UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS

Ciências
ULisboa

Vulnerability Detection in Device Drivers

Doutoramento em Informática
Especialidade de Ciência da Computação

Manuel José Ferreira Carneiro Mendonça

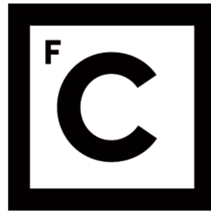
Tese orientada por:

Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Documento especialmente elaborado para a obtenção do grau de doutor

2017

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS



Ciências
ULisboa

Vulnerability Detection in Device Drivers

Doutoramento em Informática
Especialidade de Ciência da Computação

Manuel José Ferreira Carneiro Mendonça

Tese orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Júri:
Presidente:
- Doutor Luís Manuel Pinto da Rocha Carriço

Vogais:
- Doutor Marco Paulo Amorim Vieira
- Doutor André Ventura da Cruz Marnoto Zuquete
- Doutor Nuno Fuentecilla Maia Ferreira Neves
- Doutor António Casimiro Ferreira da Costa
- Doutora Ana Paula Boler Cláudio

Documento especialmente elaborado para a obtenção do grau de doutor

2017

Abstract

The constant evolution in electronics lets new equipment/devices to be regularly made available on the market, which has led to the situation where common operating systems (OS) include many device drivers (DD) produced by very diverse manufactures. Experience has shown that the development of DD is error prone, as a majority of the OS crashes can be attributed to flaws in their implementation.

This thesis addresses the challenge of designing methodologies and tools to facilitate the detection of flaws in DD, contributing to decrease the errors in this kind of software, their impact in the OS stability, and the security threats caused by them. This is especially relevant because it can help developers to improve the quality of drivers during their implementation or when they are integrated into a system.

The thesis work started by assessing how DD flaws can impact the correct execution of the Windows OS. The employed approach used a statistical analysis to obtain the list of kernel functions most used by the DD, and then automatically generated synthetic drivers that introduce parameter errors when calling a kernel function, thus mimicking a faulty interaction. The experimental results showed that most targeted functions were ineffective in the defence of the incorrect parameters. A reasonable number of crashes and a small number of hangs were observed suggesting a poor error containment capability of these OS functions.

Then, we produced an architecture and a tool that supported the automatic injection of network attacks in mobile equipment (e.g., phone), with the objective of finding security flaws (or vulnerabilities) in Wi-Fi drivers. These DD were selected because they are of easy access to an external adversary, which simply needs to create malicious traffic to exploit them, and therefore the flaws in their implementation could have an important impact. Experiments with the tool uncovered a previously unknown vulnerability that causes OS hangs, when a specific value was assigned to the TIM element in the Beacon frame. The experiments also revealed a potential implementation problem of the TCP-IP stack by the use of disassociation frames when the target device was associated and authenticated with a Wi-Fi access point.

Next, we developed a tool capable of registering and instrumenting the interactions between a DD and the OS. The solution used a wrapper DD around the binary of the driver under test, enabling full control over the function calls and parameters involved in the OS-DD interface. This tool can support very diverse

testing operations, including the log of system activity and to reverse engineer the driver behaviour. Some experiments were performed with the tool, allowing to record the insights of the behaviour of the interactions between the DD and the OS, the parameter values and return values. Results also showed the ability to identify bugs in drivers, by executing tests based on the knowledge obtained from the driver's dynamics.

Our final contribution is a methodology and framework for the discovery of errors and vulnerabilities in Windows DD by resorting to the execution of the drivers in a fully emulated environment. This approach is capable of testing the drivers without requiring access to the associated hardware or the DD source code, and has a granular control over each machine instruction. Experiments performed with Off the Shelf DD confirmed a high dependency of the correctness of the parameters passed by the OS, identified the precise location and the motive of memory leaks, the existence of dormant and vulnerable code.

Keywords: Device drivers; Dependability & security; Automated error detection; Emulation.

Resumo

A constante evolução da eletrónica tem como consequência a disponibilização regular no mercado de novos equipamentos/dispositivos, levando a uma situação em que os sistemas operativos (SO) mais comuns incluem uma grande quantidade de gestores de dispositivos (GD) produzidos por diversos fabricantes. A experiência tem mostrado que o desenvolvimento dos GD é sujeito a erros uma vez que a causa da maioria das paragens do SO pode ser atribuída a falhas na sua implementação.

Esta tese centra-se no desafio da criação de metodologias e ferramentas que facilitam a deteção de falhas nos GD, contribuindo para uma diminuição nos erros neste tipo de software, o seu impacto na estabilidade do SO, e as ameaças de segurança por eles causadas. Isto é especialmente relevante porque pode ajudar a melhorar a qualidade dos GD tanto na sua implementação como quando estes são integrados em sistemas.

Este trabalho inicia-se com uma avaliação de como as falhas nos GD podem levar a um funcionamento incorreto do SO Windows. A metodologia empregue usa uma análise estatística para obter a lista das funções do SO que são mais utilizadas pelos GD, e posteriormente constrói GD sintéticos que introduzem erros nos parâmetros passados durante a chamada às funções do SO, e desta forma, imita a integração duma falta. Os resultados das experiências mostraram que a maioria das funções testadas não se protege eficazmente dos parâmetros incorretos. Observou-se a ocorrência de um número razoável de paragens e um pequeno número de bloqueios, o que sugere uma pobre capacidade das funções do SO na contenção de erros.

Posteriormente, produzimos uma arquitetura e uma ferramenta que suporta a injeção automática de ataques em equipamentos móveis (e.g., telemóveis), com o objetivo de encontrar falhas de segurança (ou vulnerabilidades) em GD de placas de rede Wi-Fi. Estes GD foram selecionados porque são de fácil acesso a um atacante remoto, o qual apenas necessita de criar tráfego malicioso para explorar falhas na sua implementação podendo ter um impacto importante. As experiências realizadas com a ferramenta revelaram uma vulnerabilidade anteriormente desconhecida que provoca um bloqueio no SO quando é atribuído um valor específico ao campo TIM da mensagem de Beacon. As experiências também revelaram um potencial problema na implementação do protocolo TCP-IP no uso das mensagens de desassociação quando o dispositivo alvo estava associado e autenticado com o ponto de acesso Wi-Fi.

A seguir, desenvolvemos uma ferramenta com a capacidade de registar e instrumentar as interações entre os GD e o SO. A solução usa um GD que envolve o código binário do GD em teste, permitindo um controlo total sobre as chamadas a funções e aos parâmetros envolvidos na interface SO-GD. Esta ferramenta suporta diversas operações de teste, incluindo o registo da atividade do sistema e compreensão do comportamento do GD. Foram realizadas algumas experiências com esta ferramenta, permitindo o registo das interações entre o GD e o SO, os valores dos parâmetros e os valores de retorno das funções. Os resultados mostraram a capacidade de identificação de erros nos GD, através da execução de testes baseados no conhecimento da dinâmica do GD.

A nossa contribuição final é uma metodologia e uma ferramenta para a descoberta de erros e vulnerabilidades em GD Windows recorrendo à execução do GD num ambiente totalmente emulado. Esta abordagem permite testar GD sem a necessidade do respetivo hardware ou o código fonte, e possuí controlo granular sobre a execução de cada instrução máquina. As experiências realizadas com GD disponíveis comercialmente confirmaram a grande dependência que os GD têm nos parâmetros das funções do SO, e identificaram o motivo e a localização precisa de fugas de memória, a existência de código não usado e vulnerável.

Palavras-Chave: Gestores de dispositivos; Confiabilidade & segurança; Deteção automática de erros; Emulação.

Resumo Alargado

Os computadores são ferramentas comuns na vida moderna. Ao longo dos anos a arquitetura dos sistemas operativos (SO) evoluiu de forma a ser o mais independente possível do hardware, acomodando a constante evolução da tecnologia. Esta flexibilidade e extensibilidade é obtida através dos gestores de dispositivos, componentes chave do sistema que atuam como interface entre o SO e o hardware.

Devido à constante evolução da eletrónica de consumo, aparecem continuamente novos gestores de dispositivos. Paralelamente, os SO tendem a manter a compatibilidade com diferentes gerações de gestores devido à impossibilidade prática de os reescrever. Ambos aspetos contribuem para que os gestores de dispositivos sejam um dos componentes de software mais dinâmicos e em maior número nos SO atuais.

O desenvolvimento de um gestor de dispositivo é uma tarefa complexa que exige variados conhecimentos sobre a estrutura do SO e do hardware, algo que não é normalmente compreendido na sua totalidade pela maior parte dos programadores. Além disso, a manutenção e teste deste tipo de software é uma das tarefas mais onerosas na produção e manutenção dos SO.

Na maior parte dos casos os gestores de dispositivos são considerados parte integrante do SO, e como tal, um erro neste tipo de software normalmente traz consequências catastróficas para o sistema. No entanto, muitos dos utilizadores e administradores de sistemas arrisca a instalação de gestores de dispositivos sem verificação prévia da sua confiabilidade. Estas razões levam a que os gestores de dispositivos apareçam como uma das principais causas na falha dos sistemas, devido à existência de erros de implementação.

O teste de software é um dos principais mecanismos na descoberta de erros. Todavia, a procura de erros em aplicações e hardware é um processo minucioso e demorado que, dada a complexidade dos sistemas de hoje em dia, se torna bastante difícil de ser realizado por seres humanos. Assim tem-se recorrido à automatização dos processos de teste, recorrendo a técnicas de análise automática do código ou de injeção de faltas durante o processo de implementação. No entanto, no caso dos gestores de dispositivos, a tarefa de procura de erros é dificultada pelo facto de que na maioria dos casos este tipo de software é disponibilizado sem acesso ao código fonte. Além disso a estimulação do código do

gestor de dispositivo requer normalmente a montagem dum sistema de testes com alguma complexidade.

Este trabalho centra-se nos desafios relacionados com a deteção de erros em gestores de dispositivos, desejando contribuir para a redução de erros neste tipo de software, do seu impacto na estabilidade do SO e das ameaças de segurança causadas pela sua exploração por agentes maliciosos. Isto torna-se especialmente relevante porque permite aos programadores melhorar a qualidade dos gestores de dispositivos durante o seu desenvolvimento ou quando estes são integrados no SO. O trabalho visa contribuir com diferentes abordagens na identificação e localização de erros, sabendo de antemão que a construção deste tipo de soluções requer que se ultrapassem várias dificuldades. Houve um enfoque no Windows por ser um dos SO mais utilizados, e por trazer desafios adicionais devido à típica inacessibilidade ao código fonte dos seus componentes, funções e gestores de dispositivos.

Numa fase inicial do trabalho pretendeu-se perceber o nível de resiliência do Windows quanto à passagem de parâmetros incorretos às funções que o SO disponibiliza aos gestores de dispositivos. A abordagem utilizou uma análise estatística para a elaboração duma lista das funções mais utilizadas pelos gestores de dispositivos presentes no SO. Essa informação foi empregue na geração de forma automática de um conjunto de gestores de dispositivos sintéticos que introduzem parâmetros incorretos nas chamadas a essas funções do SO, imitando desta forma uma falta na chamada à função. A análise dos resultados permitiu determinar quais das funções testadas eram as mais vulneráveis aos erros nos parâmetros, quais as consequências em termos de integridade do SO, nomeadamente no sistema de ficheiros, assim como a capacidade do SO em identificar a causa das paragens e bloqueios (quando existiram).

Numa outra fase deste trabalho procedeu-se ao desenvolvimento de uma metodologia e ferramenta para a injeção de ataques em gestores de dispositivos de comunicação sem fios (Wi-Fi). Uma vez que o hardware de comunicações e os seus gestores estão diretamente expostos ao meio de transmissão, violações no protocolo de comunicação são primariamente processadas por este tipo de software. A exequibilidade desta técnica de injeção depende da capacidade de manipulação do conteúdo de todos os campos das mensagens, uma vez que muitos deles são utilizados na manutenção da integridade do estado do protocolo. A arquitetura desenvolvida envolveu a automatização do desenho dos casos de teste, recorrendo a uma técnica de fuzzing para determinar os valores a utilizar em cada

campo das mensagens. Adicionalmente, procedeu-se à automatização do processo de execução dos casos de teste e recolha de resultados.

As experiências executadas com gestores de dispositivos da rede Wi-Fi demonstraram vulnerabilidades face à violação da especificação do protocolo, permitindo determinar quais os valores, campos e em que estado do protocolo era possível gerar situações de bloqueio do SO.

Apesar do sucesso demonstrado pelos resultados alcançados, o sistema anterior não era capaz de determinar com exatidão a localização do erro no código do gestor de dispositivo ou o motivo pelo qual este acontecia.

Na seguinte fase do trabalho desenhou-se a ferramenta Intercept para registar todas as interações existentes entre o SO e o gestor de dispositivo. Na sua essência, o Intercept usa um gestor de dispositivo envelope capaz de envolver, em tempo de execução, o código binário de um gestor de dispositivo alvo. Desta forma o gestor de dispositivo alvo nunca interage diretamente com o SO, e todas as funções chamadas a partir do SO ou pelo gestor de dispositivo podem ser intercetadas pelo gestor envelope. Esta técnica permitiu-nos registar e interpretar os dados envolvidos nas interações entre o gestor de dispositivo e o SO, permitindo atividades como a análise reversa do código binário, e a determinação de alguns erros nos gestores de dispositivos.

Na última fase do nosso trabalho, desenvolvemos uma metodologia que permite a localização de erros em gestores de dispositivos sem recurso ao código fonte ou a hardware específico. A metodologia assenta na ideia de que a estrutura de um gestor de dispositivo difere substancialmente da estrutura de uma aplicação. Na estrutura atual do Windows, o gestor de dispositivo regista funções no SO que obedecem a uma especificação pré-determinada, e a partir das quais o SO solicita a realização de serviços ao gestor. Por outro lado, o gestor de dispositivos utiliza um conjunto de rotinas do SO, por exemplo, para obter e libertar recursos, para interagir com o hardware, ou para manipular cadeias de valores. Além disso, existe uma sequência lógica na forma como o SO evoca as funções do gestor, desde o seu carregamento na memória até à sua terminação. Esta estrutura permite assumir, entre outras coisas, que o gestor dispositivo disponibiliza vários pontos de entrada com propósitos bem definidos, e limita o tipo de interação que o SO pode ter com o gestor. Existem vários outros aspetos que se devem verificar, incluindo a execução célere das funções disponibilizadas ao SO; a validação dos valores devolvidos pelo SO, a circunscrição aos recursos disponibilizados pelo SO (e.g., memória e identificadores de recursos) assim como a utilização dos recursos na

sequência e momentos apropriados. Como resultado destes pressupostos, é possível construir um sistema que, imitando o comportamento SO, consegue de forma controlada e sistemática estimular o gestor de dispositivos de forma a tornar evidente potenciais erros.

A ferramenta Discovery realiza esta metodologia recorrendo à emulação da execução do código binário do gestor de dispositivo, de forma a ultrapassar os constrangimentos da ausência do código fonte. Para além disso, usufrui das vantagens de realizar este tipo de análise sem necessidade de hardware especifico, assim como ter a capacidade de determinar a localização exata dos erros e as suas manifestações. Ela define um conjunto de validadores, algum deles com a granularidade de uma instrução máquina, permitindo a descoberta de erros ao mais baixo nível. Um outro conjunto de validadores garante a identificação de erros nas chamadas às funções do SO. Finalmente, um terceiro conjunto de validadores consegue aferir desequilíbrios nos recursos do SO (e.g., memória não devolvida ao SO) e encontrar código que não é executado. Os testes realizados com alguns gestores de dispositivos disponíveis comercialmente permitiram identificar algumas situações de erro, código não usado e vulnerável que demonstram o potencial deste tipo de ferramenta.

Acknowledgements

The research, development and writing of this work was performed at the same time that I was keeping a full-time demanding job in an International company. Being a worker-student, I've jumped with my laptops and research equipment from hotel to hotel, between Algeria, Mozambique, Nigeria, Portugal and Poland in the past few years.

As it is possible to understand, I must give big thanks to a few, but very important people that without their help, but mostly their support, this work would not have been possible.

First and foremost, I want to recognize my advisor, Professor Nuno Ferreira Neves, for his vision, knowledge, believe and constant support even though I was far away. Friends and colleagues at my work, Rui Meneses for giving me the opportunity for being his "right arm" but also allowing me to fly away when it came the time. Alexandre Freire, Domingos Mateus, Jerónimo Ferreira and Ricardo Godinho for keeping me part of the band while abroad. Antonio Botelho, Gonçalo Veras, Grant Ezeronye and José Martins for enduring my bad moods and be left aside while writing my thesis in foreign countries – together we have crossed a long way.

This document is dedicated to my young daughters, Ana Rita and Mariana, for their love and brave heart, for being able to keep their tears away from me while speaking through Skype, but specially as proof of not giving up. For my parents and sister for their proud and support, but above all, to my wife Susana Mendonça for loving me, being the pillar that sustained our family while I was away, supporting my dreams, keeping always a smile for me when we were face to face in different computer screens and receiving me with wide arms open every time I arrived home.

This work was my desert crossing!

*"Space Shuttle **Discovery** (…) is one of the orbiters from NASA's Space Shuttle program and the third of five built. (…). Over 27 years of service it launched and landed 39 times, gathering more spaceflights than any other spacecraft to date."*

Space Shuttle Discovery in Wikipedia, June 2016

"We choose to go to the moon in this decade and do the other things, not because they are easy, but because they are hard…"

John F. Kennedy 12 of September 1962

To my family, with all my love.

CONTENTS

LIST OF FIGURES xv
LIST OF TABLES xvii
LIST OF ALGORITHMS AND LISTINGS xix
LIST OF ACRONYMS xxi
LIST OF PUBLICATIONS xxv
CHAPTER 1 INTRODUCTION 1
1.1 The Inherent Complexity of DDs 2
1.2 Objective and Overview of the Work 4
1.3 Structure of the Thesis 6
CHAPTER 2 DEVICE DRIVERS 9
2.1 Introduction 10
2.2 Windows Device Drivers 12
2.3 Linux Modules 16
2.4 Microkernels 20
2.5 Microdrivers 22
2.6 Virtual Machines 24
2.7 Summary 29
CHAPTER 3 RELATED WORK 33
3.1 Preparatory Concepts 34
3.2 Fault Injection 37

3.3 Robustness Testing 47
3.4 Instrumentation and Dynamic Analysis 50
3.5 Isolation of Device Driver Execution 52
3.6 Static Analysis 56
3.7 Driver Programming Model 61
3.8 Summary 65
CHAPTER 4 ROBUSTNESS TESTING OF THE WINDOWS DRIVER KIT 69
4.1 The Test Methodology 70
4.2 Selecting the Candidate List 71
4.3 Tested Faulty Values 74
4.4 Expected Failure Modes 75
4.5 Experimental Setup 76
4.6 Discussion of Results 77
4.7 Summary 86
CHAPTER 5 ATTACKING WI-FI DRIVERS 87
5.1 Wdev-Fuzzer Architecture 88
5.2 Using Wdev-Fuzzer in 802.11 90
5.3 Tested Faulty Values 92
5.4 Tested Scenarios 93
5.5 Expected Failure Modes 93
5.6 The Testing Infra-structure 94
5.7 Experimental Results 97
5.8 Summary 102
CHAPTER 6 INTERCEPT 103
6.1 Intercept Architecture 104
6.2 Using Intercept 105
6.3 Tracing the Execution of the DUT 107
6.4 Experimental Results 108
6.5 Summary 117
CHAPTER 7 SUPERVISED EMULATION ANALYSYS 119
7.1 Methodology 120
7.2 Assumptions on Device Driver Structure 121
7.3 Device Driver Flaw Classes 123
7.4 Detecting Flaws with Validators 124
7.5 Platform Architecture 126
7.6 Procedures 128

7.7 Discovery Framework 131
7.8 Discovery Emulation Execution Mechanisms 140
7.9 Detection of Flaws... 142
7.10 Experimental Results 150
7.11 Summary 167
CHAPTER 8 CONCLUSIONS AND FUTURE WORK 169
8.1 Conclusions... 169
8.2 Future Work... 171
ANNEX I – Robustness Testing of the Windows DDK sample code 173
ANNEX II – Discovery 183
BIBLIOGRAPHY 187

LIST OF FIGURES

Figure 2-1: Xen architecture. 26
Figure 2-2: Xen network driver organization. 27
Figure 2-3: VMware architecture. 28
Figure 3-1: Basic components of a fault injection system. 46
Figure 4-1: Test DD generation. 71
Figure 4-2: Experimental setup. 77
Figure 4-3: Relative robustness (FM1/#DD). 79
Figure 4-4: File System sensitiveness (FM4/(FM3+FM4)). 83
Figure 4-5: Source identification OK (M1). 84
Figure 4-6: Source identification error (M2). 85
Figure 4-7: Source of crash unidentified (M3). 85
Figure 5-1: Wdev-Fuzzer block diagram. 89
Figure 5-2: Generic Wi-Fi MAC frame format. 90
Figure 5-3: Relationship between messages and services in Wi-Fi. 91
Figure 5-4: Fuzzer Wi-Fi test infrastructure. 95
Figure 6-1: Intercept architecture. 105
Figure 6-2: Drive initialization – Call to NdisMRegisterMiniportDriver. 110
Figure 6-3: Call to MPInitializeEx to initialize the hardware (excerpt). 111

Figure 6-4: Call to NdisGetBusData / SetBusData. 112
Figure 6-5: Looking in detail at a particular packet (excerpt). 113
Figure 6-6: DD disabling process (excerpt). 114
Figure 7-1: Discovery Framework Architecture. 132
Figure 7-2: Example of Discovery Memory organization. 137
Figure AnII-1: Discovery Main Window (general view). 183
Figure AnII-2: Discovery5 Console. 184
Figure AnII-3: DCPU and integrated debugger windows. 184
Figure AnII-4: Example of Driver Entry Call Graph (pre-Expanded) 185
Figure AnII-5: Example of Driver Entry Call Graph (Expanded) 186
Figure AnII-6: Dynamic Report 186

LIST OF TABLES

Table 4-1: Drivers in a Windows OS installation. 72
Table 4-2: Top 20 called DDK functions. 73
Table 4-3: Top 20 functions driver coverage. 73
Table 4-4: Fault type description. 74
Table 4-5: Expected failure modes. 75
Table 4-6: Observed failure modes. 78
Table 4-7 Return error (RErr) values. 81
Table 4-8 Return OK (ROk) values. 82
Table 5-1: Tested Wi-Fi frames. 91
Table 5-2: Tested Faulty Values. 92
Table 5-3: Expected failure modes. 94
Table 5-4: Detailed F1 failure mode. 94
Table 5-5: Observed Failure Modes in Scenario A. 99
Table 5-6: Observed Failure Modes in Scenario B and C. 101
Table 6-1: Device drivers under test. 108
Table 6-2: Average file transfer time and speed values. 109
Table 6-3: Statistics of resource allocation/deallocation. 114

Table 6-4: Statistics of resource allocation/deallocation. 114
Table 6-5: Top 5 most used functions by each driver. 115
Table 7-1: Summary of Implemented Windows Emulator functions. 138
Table 7-2: List of Primitive Checkers. 143
Table 7-3: List of implemented validators and detectable flaws. 144
Table 7-4: DITC test values. 146
Table 7-5: IFTC combination values. 147
Table 7-6: Applicable call sequence test conditions (not exhaustive). 148
Table 7-7: Expected failure modes. 149
Table 7-8: Count of Lines of Code of Discovery. 150
Table 7-9: Characteristics of the BT DD. 151
Table 7-10: Imported functions test cases for BT. 152
Table 7-11: Discarded import functions test cases for BT (not exhaustive). 153
Table 7-12: BT Driver Interface and test cases. 153
Table 7-13: Example Test Set for BT and execution time. 154
Table 7-14: Detail of BT_TS1 Test Set 155
Table 7-15: Test Results for BT. 156
Table 7-16: Relation between the test sets and the identified errors. 157
Table 7-17: Characteristics of the SR DD. 158
Table 7-18: SR imported functions test cases. 159
Table 7-19: SR Driver Interface and test cases. 160
Table 7-20: SR Driver Interface and test cases (continued). 161
Table 7-21: Test Set for SR. 162
Table 7-22: Test Set for SR (continued). 163
Table 7-23: Test Results for SR. 164
Table 7-24: Execution time of Discovery during the loading process. 165
Table 7-25: Performance of Discovery during execution. 166

LIST OF ALGORITHMS AND LISTINGS

List 2-1: DriverEntry prototype. 14
List 2-2: DRIVER_OBJECT definition (subset). 14
List 2-3: Struct device in Linux (sample). 17
List 2-4: Functions to register and unregister devices. 18
List 2-5: Struct device_driver in Linux (subset). 18
List 2-6: Functions to register and unregister DDs 19
List 7-1: TDiscoveryMemory definition (sample). 134
List 7-2: DCPU structure (sample). 135
List 7-3: TFuncTranslation – Linkage of imported functions. 139

LIST OF ACRONYMS

AP	Access Point
API	Application Programming Interface
BIOS	Basic Input/Output System
BSS	Basic Service Set
CWE	Common Weakness Enumeration
DD	DD
DDK	DD Kit
DFI	Dynamic Fault Injection
DMA	Direct Memory Access
DPI	Driver Programming Interface
DS	Distribution System
ELF	Executable and Linkable Format
EMI	Electro-magnetic interference

FIS Fault Injection System

FL Fixed Length

FPGA Field Programmable Gate Array

FT Fault Tree

FTP File Transfer Protocol

GUI Graphical User Interface

HDL Hardware Description Language

HID Human Interface Device

IBSS Independent BSS

IC Integrated Circuit

ICs Integrated Circuits

IRQL Interrupt Request Level

IDE Integrated Drive Electronics

IMM Integrated Memory Management

I/O Input/Output

IP Internet Protocol

IPC Inter-process Communication

IRP I/O Request Packet

IRQ Interrupt Request

IRQL Interrupt Request Level

ISR Interrupt Service Routine

LLVM Low Level Virtual Machine

MAC Media Access Controller

MC Markov Chain

MMU Memory Management Unit

MTU Maximum Transmission Unit

NAT	Network Address Translation
NDIS	Network Driver Interface Specification
NIC	Network Interface Card
OS	Operating System
PCC	Proof-Carrying Code
PCI	Peripheral Component Interconnect
PEF	Portable Executable File Format
PnP	Plug and Play
POSIX	Portable Operating System Interface
RPC	Remote Procedure Call
RTL	Register Transfer Level
SCSI	Small Computer System Interface
SMP	Symmetric Multi-Processing
STA	Member Station
SUT	System Under Test
SWIFI	Software Implemented Fault Injection
TAP	Test Access Ports
TCP	Transmission Control Protocol
TLV	Tag Length Value
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration
VM	Virtual Machine
VMM	Virtual Machine Monitor
WDD	Windows DD

WDM Windows Driver Model

WLAN Wireless Local Area Network

LIST OF PUBLICATIONS

International Conferences

[P1] *“Robustness Testing of the Windows DDK”*, Manuel Mendonça and Nuno Neves, In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 554-563, June 2007

[P2] *“Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities”*, Manuel Mendonça and Nuno Neves, In Proceedings of the Seventh European Dependable Computing Conference, pp. 110-119, May 2008

[P3] *“Intercept – Profiling Windows Network Device Drivers”,* Manuel Mendonça and Nuno Neves, In Proceedings of the 14th European Workshop on Dependable Computing, pp. 61-75, May 2013

National Conferences

[P4] *“Testes de Robustez ao DDK do Windows XP”*, Manuel Mendonça e Nuno Neves, Actas da 2ª Conferência Nacional Sobre Segurança Informática nas Organizações (SINO06), Aveiro, Portugal, October, 2006.

[P5] *"Localização de Vulnerabilidades de Segurança em Gestores de Dispositivos Wi-Fi com Técnicas de Fuzzing"*, Manuel Mendonça e Nuno Neves, Actas da 3ª Conferência Nacional Sobre Segurança Informática nas Organizações (SINO07), Lisboa, Portugal, October, 2007.

Waiting submission result

[P6] "Discovery – Finding vulnerabilities in windows DDs without source code", Manuel Mendonça and Nuno Neves, manuscript.

CHAPTER 1 INTRODUCTION

Computers are common tools in modern life. In their short history, they have suffered huge improvements achieving a very important role in our society, being used in a wide variety of activities ranging from work to leisure.

Over the years, operating systems (OS) evolved their architectures to become, as much as possible, independent from hardware in order to accommodate the constant evolution of motherboards and connected devices. Their flexibility and extensibility is achieved by the inclusion of device drivers (DD), which act as the interface between the OS and the hardware.

Given the typical short life cycle of consumer electronics, system designers have to constantly program new drivers. In parallel, OS developers have to maintain compatibility with legacy DD, as it is practically impossible to rewrite them for a new architecture, given that their design is normally dependent on low level details. To accommodate the large number of devices that can be connected to a computer [1], it is usually possible to find thousands of drivers included in an OS installation. These aspects contribute to make DD the most dynamic and largest part of the OS nowadays.

Even though current drivers are mostly written in a high level language (e.g., C or C++), they continue to be difficult to build and verify. Their development requires knowledge from a set of disparate areas, including Integrated Circuits (ICs), OS interfaces, compilers, and timing requirements, to name a few, which are often not mastered simultaneously by programmers. In addition, maintaining such wide variety of hardware makes DD development, maintenance and testing a very expensive task.

Due to the above factors, it is not surprising that drivers can contain flaws in their implementation. In some drivers, this can be particularly worrisome. For example, in DD dedicated to assist communication hardware, errors may be remotely exploited. In addition, users normally accept the installation of DDs without checking their reliability, given that they are necessary to solve an immediate problem (e.g., being able to use a certain device for which no driver was provided). Moreover, almost any flaw in DD has a catastrophic impact because they run in the OS kernel. Consequently, despite the efforts performed by both free and commercial OS organizations, DD have been traditionally one of the most important causes of failures in popular systems, such as Linux [2][3] and Windows [5].

It should be possible to design tools to identify errors in drivers, which users and system administrators could rely on to evaluate DDs. However, the growing complexity of both hardware and software tends to make the evaluation of dependability attributes a hard task. The use of an analytical model is even more difficult as the mechanisms involved in the fault activation and error propagation are quite intricate and may not be completely understood. In order to make the analysis feasible, sometimes simplifying assumptions have to be employed, with the cost of reducing the applicability of the final results.

1.1 The Inherent Complexity of DDs

In most commodity OS, such as Windows and Linux, DD are passive objects build as a collection of entry points that are invoked by the kernel when it needs a particular service. The driver executes in the context of external OS threads. Even if the driver creates one or more threads to handle auxiliary tasks, the driver logic is invoked from the OS. This model enables the kernel to efficiently communicate with the driver by invoking function calls, but it complicates driver programming as it needs to be designed to handle multiple concurrent executions. DD are state-full objects whose reaction to a request depends on the history of previous requests and

replies. Thus, a driver must maintain its execution state across invocations using state variables, which need to be stored in memory regions requested from the OS. Additional constraints in timings and non-blocking further complicates the management of the concurrency.

The development of DDs is nowadays performed using high-level languages such as C or C++. The set of header files, source-code and other libraries requires multiple files to be maintained, which can lead to complex makefiles (or projects). In monolithic designs, such as in commodity OS, all the kernel functions run in privileged mode. DD are extensions of the kernel code and they can perform direct memory access operations (i.e., they can write in arbitrary locations of physical memory), including over kernel data structures. Therefore, any bug in a DD can potentially corrupt the entire system.

Debugging and testing a DD requires often the associated piece of hardware to be present and to be responsive. The complexity associated with DD's testing is aggravated as most vendors do not release openly the hardware specification. When they do, the specification many times contains inaccuracies and errors. In most OS, the debugging and testing tasks usually involves the use of two machines where one runs the debugger and the other is the target system where the driver is executed. Often many hours of work are needed just to setup this debugging environment. The debugging process itself is mostly done using a trial and error approach, setting up break points and conditions that make the driver fail, and restarting the target machine each time it hangs or crashes.

Maintaining driver code is also an issue. Due to the difficulties in driver development, many times the code is adapted to new OS versions without taking into consideration the novel features, which would recommend significant rework to be performed. Sometimes there are even changes in the new OS versions that do not maintain retro compatibility.

DD are complex to code, to debug and to maintain, and therefore are viewed by even experienced programmers, system administrators and users as an obscure and complex section of the OS. Over the year's various initiatives and tools have been created to assist in DD testing (many of them are reviewed later on in this document). However, even in carefully tested DD, often it is still possible to find flaws.

1.2 Objective and Overview of the Work

This thesis is primarily motivated by the existence of errors in DD, their impact in the OS stability and the security threats that these flaws may represent. It aims on one side to assess how a faulty driver can compromise the correct execution of an OS, and on the other side to develop mechanisms capable of discovering DD flaws. This is especially relevant because it can help developers to build DD that operate in a more dependable manner. Users and system administrators can also benefit from such tools to both evaluate existing systems or before doing upgrades.

In our approach, we assume that all interactions with other drivers and applications are performed with the OS acting as an intermediary. The detection of DD flaws is performed mainly using techniques that do not require access to the driver source code. Our solutions use as input the binary image of the DD and output the set of problems that were identified. We are especially interested in supporting systems where the source code is not available because it makes our solutions applicable to a wider set of testing scenarios. We have chosen to focus the work on Windows as it is one of the most widely used OS and in the majority of the cases the source of the DD is not available.

We started the work with an investigation with the aim to understand how DD flaws can impact the correct execution of an OS. To accomplish this, we have performed a statistical analysis of the DD that exist in a Windows installation, and then we have obtained the list of the most used OS functions. Next, we have developed a mechanism that automatically builds DD and injects faults when those drivers make function calls to the kernel. This approach differs from other robustness tests performed in the past (see for instance [53][54]), in the sense that it does not use an existing DD to insert the faults. Since we use synthetic drivers, our approach ensures that the fault is always activated. The obtained results confirmed that a DD can cause serious damage to the OS only by calling functions with invalid arguments and provided insights of the most common DD bugs.

Secondly, we researched how to externally attack a DD. For this purpose, we have developed the Wdev-Fuzzer architecture. Although it was built for Wi-Fi networks, Wdev-Fuzzer can be easily adapted to other wireless network technologies. The methodology consisted in injecting potential erroneous values in the fields of the Wi-Fi frames, thus simulating an external attack. This allowed an evaluation of the behaviour of a target system in the presence of frames that violate the Wi-Fi specification. Our experiments with an HP PDA device revealed the

existence of several types of problems, showing that this device could be successfully attacked by a remote adversary.

Next, we wanted to understand the type of interactions that exist between a DD and the OS. We developed a technique to control and interfere in the binary execution of the DD under test (DDUT). For this, we have developed Intercept, a system that wraps the execution of a Windows DD. A wrapper DD (WRDD) is used to provide an execution environment for the DDUT, supporting the load of the DDUT binary image into the address space of the WRDD and dynamically linking the DDUT to the OS. The WRDD mediates all interactions between the DDUT and the OS and is capable of recording the exchanged information and interfere with them. The information collected by Intercept documents and clarifies the correct order of the function calls, the parameters contents and return values. Additionally, Intercept maintains statistical information of several OS objects usage, such as memory allocation/deallocation and spinlocks. This type of information is useful in debugging and reverse engineering the DD and OS. The interference capability of Intercept supports the modification of parameters and return values passed in the function calls (from the OS to the DDUT and vice-versa). This was used to test the DDUT, but the likelihood of hanging or crashing the system is very high since an incorrect parameter or return value could corrupt the kernel.

The results showed the profiling capability to inspect network traffic by accessing with Intercept the data packets available in function parameters. It helped to understand complex interactions with the OS, clarifying for instance the order of their execution. Statistics maintained by Intercept helped to evaluate resource usage and potential resource leakages during the DD activity. Using the interference capabilities of Intercept, it was possible to test the behaviour of a Wi-Fi DD when incorrect parameters are passed by the OS (in a simulated environment) and uncover an incorrect order of parameter validations.

To overcome the difficulties related with the absence of the DD source code and associated hardware, we have designed the Supervised Emulation Analysis methodology. The methodology uses emulation with granular control over the machine instructions and a set of validators capable of capturing low level errors. Another set of validators acts whenever the DDUT calls an (emulated) OS function to check the parameters against several constrains. A test manager stimulates the DDUT at the exposed interfaces, mimicking the OS and controlling the return and parameter values of the OS functions as well as the different DDUT code paths. Tests performed with some off the shelf Windows DD confirmed the feasibility of the

methodology and the capability in capturing DD errors (e.g., memory leaks) and finding dormant and vulnerable code.

1.3 Structure of the Thesis

This thesis is organized as follows.

Chapter 2 provides an overview of the device driver organization. We start by briefly describing Windows and Linux drivers to explain their structure and relationship with the OS. The chapter continues by making references to microkernels to get some insights on other alternative solutions. Microdrivers give us another approach with the benefit of reducing the effects of faulty DD. The chapter concludes with virtual machines, focusing on understanding how they address the isolation of DD.

Chapter 3 is dedicated to the related work. It starts by providing some introductory concepts and describes some of the key research areas to which this thesis relates, such as fault injection, robustness testing, instrumentation, static and dynamic analysis. It helps to understand some of the decisions taken during our implementations as well as how the developed works position it in terms of contributions.

Chapter 4 describes a solution for testing the Windows OS and its interfaces through the Windows DD Kit. We present a novel technique to automatically build test campaigns taking as input an XML description of the Windows functions. The result of this research contributed to understand how the Windows OS handles faulty DD, what are the main causes for the observed hangs and crashes, and the effects on the file system.

Chapter 5 addresses attacks on Wi-Fi drivers using a new fuzzer architecture that is able to build malformed packets and execute test cases against a target system. The results revealed some disturbing conclusions over the possibility of causing crashes in remote machines just by sending malformed packets with the Wi-Fi protocol.

Chapter 6 takes us deeper in the interactions between the OS kernel and the DD, and presents some of the necessary techniques to build a layer that can stand in between these two components. Intercept is the resulting tool supporting the discovery of flaws in DDs.

Chapter 7 presents a methodology and framework that enable researchers to locate errors and vulnerabilities in DD through the emulation of the OS and hardware. We present the results obtained with some off the shelf Windows DD.

We conclude the thesis in chapter 8 with a summary of the investigation and a description of future work.

CHAPTER 2 DEVICE DRIVERS

Nowadays the three most used operating systems for personal computers are Windows (88%), OSX (4%) and Linux (2%) [4]. However, OSX is a proprietary operating system based in the Open Darwin Unix, thus having the same roots as Linux. On the emerging market of mobile phones, tablets and other similar devices, the share is around: Android (69%), iOS (26%) and Windows Phone (2%). Since iOS is based on Open Darwin (Unix) and Android has its origins in Linux kernel 6 the same is to say that both platforms are based on Unix like systems. This justifies the argument that nowadays Windows and Linux/Unix constitute the two major families of devices drivers in the computer industry. There are however other approaches to OS structure. For instance, instead of placing the DD as part of the kernel code it can be implemented like any other user-space application, allowing the driver to be started and stopped just as any other program.

This chapter starts with a short section on DDs organization. We will describe what a DD is and focus the presentation on the structure and operation of DDs on the more popular operating systems (Windows and Linux). This will help to understand the internal architecture of this kind of software, its complexity and reliability issues. More detailed information about Windows DDs can be obtained in

[163][160][161][162] and for Linux, information about drivers can be found in [117][118][119].

The chapter continues with DD organization in Microkernels, as a solution to provide the necessary level of isolation between the kernel and the DDs. We will also describe Microdrivers as another proposal for isolating the DDs, while keeping the performance of the system mostly unaffected. Finally, we will address Virtual Machines as a new trend to abstract resources and how DDs play an important role on the dependability of such systems.

2.1 Introduction

A device is a hardware piece attached to the computer, such as the keyboard, a network card or a display card. A DD is operating system code that allows the computer to communicate with a specific device. A DD consists of a set of functions implementing its logic and provides services to the rest of the OS. On monolithic OS, such as Windows and Linux, DDs can access the whole set of functions of the kernel, not only those that are used to carry out operations in the kernel space, but also in the application space.

DDs can be organized in several functional classes, like Memory Management, Interrupt Management, File System Management, and Control Block Management. Two main categories of drivers can be distinguished:

- Software drivers that have no direct access to the hardware layer of the devices, but rather to an abstraction (e.g., TCP/IP stack or file system);
- Hardware drivers that interact with hardware devices, either peripheral (e.g., network, disk, printer, keyboard, mouse or screen) or internal to the motherboard (e.g., bus or RAM).

In either case the drivers provide an abstract interface for the OS to interact with the hardware and the environment. A DD can thus be considered as the lowest level of software as it is directly bound to the hardware features of the device. Each driver manages one or more pieces of hardware while the kernel handles process scheduling, interrupts, etc. The operating system kernel can be considered a software layer running on top of DDs.

Depending on the type of device, the DD can operate in two different ways. In the first one, the driver accesses the device in a periodic fashion (pooling) - the driver programs a timer with a pre-defined value and whenever the timer expires the device is checked to see if it needs servicing. In the second way, the device triggers an

interrupt to request the processor's attention. Each interrupting device is assigned an identifier called the interrupt request (IRQ) number. When the processor detects that an interrupt has been generated on an IRQ, it stops the current execution and invokes an interrupt service routine (ISR) registered by the associated driver to attend to the device. In either case, these critical pieces of code must be quickly executed to prevent the whole system from being stopped.

The communication between the driver and the device is performed through read and writes in a set of registers. These may be mapped onto the memory of the computer or use a special set of read and write functions. The rules that dictate when a register can be accessed often include specific conditions on the logical state of the device. Some registers cannot be accessed when the device interrupts are turned on. Others do not have meaningful information unless other registers are read or written first.

A register may be readable, writable or both. Registers have specific length, and each bit may have particular meanings that may change depending on a read or write operation. The number of bytes that can be written and read simultaneously depends on the physical architecture of the computer.

A readable register may have a specified set of values that might be read from it. Correspondingly, a writable register may only safely accept a specific set of values. Outside of that interval the value might cause unknown or unwanted behaviour. When a driver fails to meet the specification for its associated device, the device can be placed in an invalid state where it becomes damaged or restarts potentially causing data loss.

Typically, the kernel features three main interfaces with the environment:

- The hardware layer where interactions are made via the raising of hardware exceptions and transferring data through registers;
- The Application Programming Interface (API) where the main interactions concern the application to kernel calls and,
- The interface between the drivers and the kernel, offered by the Driver Programming Interface (DPI).

In most popular operating systems, the kernel and drivers are executed in privileged mode, whereas application processes are run in a restricted address space in non-privileged mode. This reduces the risk of an application process to corrupt the kernel address space. On the other hand, since DDs execute in kernel space, any faulty behaviour is likely to impact the operation of the system.

Programming drivers with languages (such as C) that use pointer arithmetic without Integrated Memory Management (IMM) represent a special threat because it is easy to make unnoticed mistakes that corrupt the kernel.

2.2 Windows Device Drivers

The Windows Driver Model (WDM) defines a unified approach for all kernel-mode Windows drivers. It supports a layered driver architecture in which every device is serviced by a driver stack. Each driver in this chain isolates some hardware-independent features from the drivers above and beneath it, avoiding the need for the drivers to interact directly with each other. The driver manager is in charge of automatically detecting the match between installed devices and the drivers. Moreover, it finds out the dependencies between drivers such that it is able to build the stack of drivers.

The WDM has three types of DDs, but only a few driver stacks contain all kinds:

- Bus driver – There is one bus driver for each type of bus in a machine (such as PCI, PnP and USB). Its primary responsibilities include: the identification of all devices connected to the bus; respond to plug and play events; and generically administer the devices on the bus. Typically, these DDs are provided by Microsoft;
- Function driver – It is the main driver for a device. Provides the operational interface for the device, handling the read and write operations. Function drivers are typically written by the device vendor, and they usually depend on a specific bus driver to interact with the hardware;
- Filter drivers – It is an optional driver that modifies the behaviour of a device. There are several kinds of filter drivers such as: lower-level and upper-level filter drivers that can change input/output requests to a particular device.

The WDM specifies an architecture and design procedures for several types of devices, like display, printers, and interactive input. For network drivers, the Network Driver Interface Specification (NDIS) defines the standard interface between the layered network drivers, thereby abstracting lower-level drivers that manage the hardware from upper-level drivers implementing standard network transports (e.g., the TCP protocol).

Three types of kernel-mode network drivers are supported in Windows:

- Miniport drivers - A Network Interface Card (NIC) is normally supported by a miniport driver that has two basic functions: manage the NIC hardware, including the transmission and reception of data; interface with higher-level drivers, such as protocol drivers through the NDIS library. The NDIS library encapsulates all operating system routines that a miniport driver must call (functions `NdisMXxx()` and `NdisXxx()`). The miniport driver, in turn, exports a set of entry points (`MPXxx()` routines) that NDIS calls for its own purposes or on behalf of higher-level drivers to send packets.

- Protocol Drivers - A transport protocol (e.g., TCP) is implemented as a protocol driver. At its upper edge, a protocol driver usually exports a private interface to its higher-level drivers in the protocol stack. At its lower edge, a protocol driver interfaces with miniport drivers or intermediate network drivers. A protocol driver initializes packets, copies data from the application into the packets, and sends the packets to its lower-level drivers by calling `NdisXxx()` functions. It also exports a set of entry points (`ProtocolXxx()` routines) that NDIS calls for its own purposes or on behalf of lower-level drivers to give received packets.

- Intermediate Drivers - These drivers are layered between miniport and protocol drivers, and they are used for instance to translate between different network media. An intermediate driver exports one or more virtual miniports at its upper edge. A protocol driver sends packets to a virtual miniport, which the intermediate driver propagates to an underlying miniport driver. At its lower edge, the intermediate driver appears to be a protocol driver to an underlying miniport driver. When the miniport driver indicates the arrival of packets, the intermediate driver forwards the packets up to the protocol drivers that are bound to its miniport.

Windows DD structure

Windows DDs expose functions that provide services to the OS. However, only one function is directly known by the OS, as it is the only one that is retrieved from the binary file when the driver is loaded. By convention, the function name is `DriverEntry()` and is defined as represented in List 2-1.

```
1     NTSTATUS DriverEntry(
2     _In_  struct _DRIVER_OBJECT *DriverObject,
3     _In_  PUNICODE_STRING RegistryPath
4     )
```

List 2-1: DriverEntry prototype.

This function is called when the OS finishes loading the binary code of the driver, and its role is to initialize all internal structures of the driver and hardware, and register to the OS the exported driver functions.

The `DriverObject` parameter contains the fields that `DriverEntry` must fill in order to register the functions to the OS. A subset of the `DriverObject` type parameters is represented in List 2-2.

```
1    typedef struct _DRIVER_OBJECT {
2     //Sample of the structure with several fields omitted
3     //Driver name
4     UNICODE_STRING DriverName;
5
6     //Registry support
7     PUNICODE_STRING HardwareDatabase;
8
9     //For registering the unloading function
10    PDRIVER_UNLOAD DriverUnload;
11
12    //For registering the dispatch routines
13    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION
14        + 1];
15   } DRIVER_OBJECT
```

List 2-2: DRIVER_OBJECT definition (subset).

The `DriverUnload` function from the above structure is set with the address of the function that should be called when the operating system decides to unload the driver. Typically, this routine is in charge of returning to the operating system all the resources that are held by the driver.

The `MajorFunction` field is a dispatch table consisting of an array of entry points for the driver's `DispatchXXX` routines. The array's index values are the `IRP_MJ_XXX` values representing each I/O Request Packet (IRP) major function code. Each driver must set entry points in this array for the `IRP_MJ_XXX` requests that the driver handles.

The Windows kernel sends IRP to the drivers containing the information of the desired dispatch function to be executed. The following are examples of the IRP codes and intended execution functions (not exhaustive):

- `IRP_MJ_WRITE`: Transfers data from the system to the drivers' device;
- `IRP_MJ_READ`: Transfers data from the device to the system;
- `IRP_MJ_PNP`: Plug and play support routine.

In the case of miniport drivers following the NDIS specification, the `DriverEntry()` function, in addition to what was described previously, initializes all internal structures of the driver and hardware, and calls the `NdisMRegisterMiniportDriver()` of the OS to indicate the supported driver functions. Examples of miniport driver functions that are registered in the NDIS are `MPInitialize()` and `MPSendPackets()`. The first is used to initialize NDIS structures and functions' registrations and the second is used to send packets through the NIC.

Windows DDs file structure

Windows normally organizes a DD as a group of several files. Files with the extension `.inf` contain plain text and are divided in several sections. They have relevant context data such as the identifier of the vendor of the driver, the type and the compatibility with devices, and start-up parameter values. They are used during driver installation to match devices with drivers and to find the associated `.sys` files. Files with the extension `.sys` are the binary executable images of the driver and they are loaded to memory to provide services to the OS. The binary files follow the Portable Executable File (PEF) format [62], the same format used to represent applications `.exe` and dynamic link libraries `.dll`.

The PEF file structure contains binary code and dependencies from other software modules (organized as tables). The binary code is mostly ready to be loaded into memory and run. However, since it can be placed anywhere in memory, there is the need to fix up the relative addresses of the function calls. Functions that refer to external modules are located in the imported functions table. This table contains the names of the external modules (`DLLs, .sys, .exe`), the function names and the address location in the memory of the running system. The addresses are resolved by the driver loader when it brings the driver in memory.

The driver is placed in execution by calling the `DriverEntry()` function. The address of this function is also obtained from the PEF file, and is located in the `AddressOfEntryPoint` field of the Optional Header section of the `.sys` file. Knowing this structure allows external systems to interface the driver code without having its source code, which is useful when designing solutions to discover vulnerabilities and other defects in DDs (as is our case).

2.3 Linux Modules

The Linux kernel is a Unix-like operating system kernel initially created in 1991 that rapidly accumulated developers and users, who adapted code from other free software projects for use with the new OS. The Linux kernel is released under the GNU General Public License version 2, making it free and open source software.

Linux has the ability to extend the set of features offered by the kernel at run time. Each piece of code that can be added to the kernel at runtime is called a module. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the `insmod` program and can be unlinked by the `rmmod` program.

Linux distinguishes three fundamental device types. Each module usually implements one of these types, and thus is classified as a char module, a block module, or a network module. This division is not rigid as programmers can choose to build modules implementing different drivers in a single piece of code. However, good programming practices advice that a different module should be created for each new functionality that is implemented, since decomposition is a key element of scalability and extendibility.

A character (char) device is one that can be accessed as a stream of bytes (like a file). A char driver is in charge of implementing this behaviour. Such a driver usually implements at the least the `open`, `close`, `read` and `write` system calls. Char devices are accessed by means of file system nodes, such as `/dev/tty1`. The distinguishing difference between a char device and a regular file is that in a regular file it is always possible to move back and forth, whereas most char devices are just data channels and therefore it is only possible to access them sequentially[1].

[1] There are however devices where it is possible to move back and forth. This usually applies to frame grabbers where the applications can access the whole data using `mmap` or `lseek`.

Like char devices, block devices are accessed by file system nodes in the `/dev` directory. A block device is a device that can host a file system. In most Unix systems, a block device can only handle I/O operations that transfer one or more blocks of data. However, Linux allows applications to read and write a block device like a char device permitting the transfer of any number of bytes at a time. Block and char devices only differ in the way data is managed internally by the kernel, as they have a different kernel/driver software interface.

Any network transaction is made through a device that is able to exchange data with other hosts. A network interface is in charge of sending and receiving data packets driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Network devices are, usually, designed around the transmission and receipt of packets, although many network connections are stream-oriented.

Some types of drivers work with additional layers of kernel support functions for a given device, and thus can be classified in other ways. For instance, one can talk about the USB modules, serial modules or SCSI modules.

Linux module structure

The 2.6 Linux device model provides a unified device model for the kernel, introducing abstractions that feature out commonalities from DDs. The device model is composed by different components such as `udev`, `sysfs`, `kobjects`, and device classes having effect on key kernel subsystems such as `/dev` node management, power management and system shutdown, communication with user space, hotpluging, firmware download, and module auto load.

At the lowest level, every device in a Linux system is represented by an instance of the `struct device` as represented in List 2-3.

```
1    struct device{
2      struct device *parent;
3      struct kobject kobj;
4      char bus_id[BUS_ID_SIZE];
5      struct bus_type *bus;
6      struct device_driver *driver;
7      void *driver_data;
8      void (*release)(struct device *dev);
9      /*Other fields omitted*/
10   };
```

List 2-3: Struct device in Linux (sample).

Next, we will perform a brief description of the fields of this structure for a better understanding of the device model of Linux. There are many other `struct device` fields but for simplicity they were omitted.

The device's parent represents the device to which it is attached to. In most cases a parent device is a bus or host controller. The `kobject` is a structure that represents this device and links it into the hierarchy of devices. The `bus_id[BUS_ID_SIZE]` is a string that uniquely identifies this particular device on the bus. PCI devices use the standard PCI ID format containing the domain, bus, device and function numbers. The `struct bus_type *bus` identifies the kind of bus the device sits on. The `struct device_driver *driver` is the driver that manages the device. The `void *driver_data` is a private data field that may be used by the DD and the `void (*release)(struct device *dev)` is the method that is called when the last reference to the device is removed.

At the least, the `parent`, `bus_id`, `bus`, and `release` fields must be set before the device structure can be registered. Devices are registered and unregistered using the functions device_register and device_unregister whose signatures are represented in List 2-4.

```
1    int device_register (struct device *dev);
2
3    void device_unregister(struct device *dev);
```

List 2-4: Functions to register and unregister devices.

The device model tracks all the drivers known to the system to enable the match between drivers with new devices.

A DD is defined by the structure listed in List 2-5.

```
1    struct device_driver{
2    char *name;
3    struct bus_type *bus;
4    struct list_head devices;
5    int (*probe)(struct device *dev);
6    int (*remove)(struct device *dev);
7    void (*shutdown)(struct device *dev);
8    /*Other fields omitted*/
9    };
```

List 2-5: Struct device_driver in Linux (subset).

The main fields of the struct device_driver have the following use: The name is the name of the driver that shows up in the sysfs; bus is the type of bus that this driver works with; devices is a list of all devices currently bound to this driver. The structure also contains some functions used to manage the device. For example, probe is called to query the existence of a specified device and whether this driver can work with it; remove is called when the device is removed from the system; and shutdown is called at shutdown time to inactivate the device.

Drivers are registered and unregistered using the functions listed in List 2-6.

```
1    int driver_register (struct device_driver *drv);
2
3    void driver_unregister(struct device_driver *dev);
```

List 2-6: Functions to register and unregister DDs

As an example, we are going to briefly describe how the PCI subsystem interacts with the driver model, introducing the basic concepts involved in adding and removing a driver from the system. These concepts are also applicable to all other subsystems that use the driver core to manage their drivers and devices.

A Linux DD needs to have at least a function that is called when the driver is loaded (e.g., enter_func), and another when the driver is unloaded (e.g., exit_func). During the compilation of the code, the compiler identifies the initialization function when it finds the directive module_init(init_func). Similarly, the compiler identifies the unloading function when the module_exit(exit_func) directive is processed.

The init_func() is where the DD initializes the peripherals and ties the driver to the rest of the system, by registering the functions that the DD offers to the OS in the available interfaces. The OS calls the init_func, which in turn will call the function __pci_register_driver(struct pci_driver*, struct *module, const char *mod_name) to link the driver with the system. Therefore, all PCI drivers must define a struct pci_driver variable that specifies the various functions that the PCI driver can support during the driver loading.

The struct pci_driver contains two important members used for completing the connection of the DD with the OS:

- .id_table: A structure that holds elements that the OS uses to match the identification of the vendor and device with the driver;

- `.probe`: A function called by the OS to announce to the DD that it should complete the matching process and tell to the OS if the DD can handle the PCI device. This function is called right after the registration process of the driver with the system when it finds a device that may be served by the recent register DD.

Depending on the hardware of the PCI card, the DD registers other structures with the OS for managing the device, for instance, `struct ethtool_ops` is used for the OS to control the PCI communication cards, and `struct iw_handler_def` to control the Wi-Fi structure. The OS uses other functions assigned to `struct pci_driver`, such as `open()`, `read()`, `ioctl()` and `write()`, through which it can interact with the driver in a standardized way.

Removing a driver starts when the OS calls the `exit_func`. The driver then calls the `pci_unregister_driver` operation, which merely calls the driver core function `driver_unregister`. The `driver_unregister` function handles some basic housekeeping, such as cleaning up some `sysfs` attributes that were attached to the driver's entry in the `sysfs` tree. It then contacts all devices that were attached to the driver and calls the respective release function.

2.4 Microkernels

Microkernels were developed with the idea that traditional operating system functionality, such as DDs, protocol stacks and file systems, would be implemented as a user-space program, allowing them to be executed like any other process. This would not only simplify the implementation of these services but also support performance tuning without worrying about unintended side effects. Additionally, robustness and reliability could be enhanced because these services would no longer be able to perform direct memory access operations into the OS, writing to arbitrary locations of physical memory, including over kernel data structures.

There are many good reasons for running DDs at user level, such as:

- Ease of development: If a driver is a normal user process it can be developed and debugged with well-known and common tools. On the contrary, in-kernel driver development requires a specific development and debugging environment (typically involving more than one machine). Furthermore, since in-kernel drivers can cause kernel malfunctions in unrelated kernel components, identifying the source of the fault can be much difficult.

- Maintainability: In systems like Linux, where the kernel and its internal interfaces change quickly, keeping drivers that depend on these interfaces can be a challenge due to the large number of dependencies that many DDs have [107]. A user-level API that formally isolates the driver interface reduces (or eliminates) these dependencies and at the same time would make them more portable across kernel versions. User-level drivers could also be written in any high-level language.

- Dependability: An in-kernel driver handles interrupts running on the stack of the process that was interrupted, and since it may not block, this requires very careful resource management to avoid unfairly blocking the current process or dead locking the kernel. User-level drivers run in their own context avoiding the issue of blocking in the interrupt handler and simplifying dead lock prevention. Additionally, normal OS resource management, including better control over resource consumption and protection against resource leaks, can be applied to user-level drivers. Hence, user-level drivers have the potential to improve system reliability. Moreover, in case of problems, a system may be able to survive a crashed user-level driver as the arguments made in favour of recursive restart apply to user-level drivers [108].

- Portability: In-kernel drivers have to be compiled for a particular kernel; when the kernel is updated, the end-user has to either recompile the driver (in case of Linux where source code is more likely to be available) or obtain a new one from the vendor. If the driver is in user space, it depends only on the user-driver API and so the same driver binary can continue to be used.

Mach [104][105] and Chorus [106] are two early examples of microkernel systems to take this approach.

MINIX3 [69][70] wanted to mitigate systems crashes due to buggy DDs, through the design and implementation of a fully compartmentalized operating system. The approach was to reduce the kernel to an absolute minimum and running each driver as a separate, unprivileged user-mode process.

The microkernel of MINIX3 is responsible for the low-level and privileged operations such as programming the CPU and MMU, handling the interrupts and perform inter-process communication. Servers provide the file system, process management and memory management functionalities. A database server is used to keep information about system processes with publish-subscribe functionalities. System processes can use it to store some data privately, for example, a restarting

system service can request state that it lost when it crashed. The system also contains a server that keeps track of all other servers and drivers running that can transparently repair the system when certain failures occur. System calls are transparently targeted to the right server by the system libraries.

The publish-subscribe mechanism decouples producers and consumers. A producer can publish data with an associated identifier. A consumer can subscribe to selected events by specifying the identifiers or regular expressions it is interested in. Whenever a piece of data is updated it automatically broadcasts notifications to all dependent components.

On top of the kernel a POSIX-conformant multi-server operating system was implemented. All servers and drivers run as independent user-mode processes and are highly restricted in what they can do, just like ordinary user applications. The servers and drivers can cooperate using the kernel's Inter-Process Communication (IPC) primitives to provide the functionality of an ordinary UNIX operating system.

Several drivers were implemented running as an independent user-mode process to prevent faults from spreading and make it easy to replace a failing driver without a reboot. Although not all driver bugs can be cured by restarting the failing driver, the authors of MINIX3 assume that the majority of driver bugs are related with timing and memory leaks for which a restart is usually enough.

Although conceptually microkernels are to provide the necessary level of isolation, it can come with the price of performance degradation and difficulties in porting the approach to other architectures. None of the early attempts to run drivers outside of the kernel, as unprivileged user code, has made a lasting impact. Therefore, user-level drivers remain an exception in conventional systems and used only for devices where performance is not critical or where the number of context switches is small compared with the work that it does (for instance, the Linux X server or some printer drivers in Windows, to name a few).

2.5 Microdrivers

The common approach taken by commodity monolithic operating systems is for the kernel to execute in privileged mode, controlling all system resources and isolating them from the user application behaviour.

Traditionally, DDs have been implemented as part of these kernels and there are many reasons that justify this approach: they had full access to all system resources which typically simplifies implementation and minimizes overhead; use of the

operating system mechanisms for multitasking, synchronization, memory management, I/O transfer and others. However, as previously described, this model allows a fault in a DD to potentially crash the whole system.

Taking into consideration that most driver code moves data between memory and an external device it is possible to partition the DD. In the Microdrivers architecture [71], a DD is split into a kernel-level k-driver and a user-level u-driver. Critical path code, such as I/O, and high-priority functions, such as interrupt handling, are implemented in the k-driver. This code enjoys the full speed of a purely kernel driver. The remaining code, which is invoked infrequently, is implemented in the u-driver and executes outside the kernel in a user-mode process. When necessary, the k-driver may invoke the u-driver.

The authors propose the use of a tool, DriverSlicer, to transform existing drivers into a Microdriver architecture. In the first phase, the tool partitions an existing code such that performance critical functions remain in the kernel. The split aims to minimize the cost of moving data and control along the performance critical path. Rarely used functions, such as those for start-up, shutdown and device configuration are relegated to the u-driver.

The slicing operation is performed using as input a programmer-supplied interface specification to identify the set of critical root functions for the driver. These are driver entry points that must execute in the kernel and include high priority functions or functions called along the data path. Because these functions typically have a standard prototype, the programmer supplies interface specifications as type signatures. The splitter automatically marks functions that match these type signatures as critical root functions.

In the second phase Driver Slicer uses the output of the splitter where each node of a call graph is marked kernel or user, based upon whether the corresponding function must execute in the k-driver or the u-driver. The code generator identifies interface functions and generates code to transfer control. An interface function is a function marked user that can potentially be called by a function marked kernel, or vice-versa. Non-interface functions are never called by functions implemented on the other side of the split, and thus does not need stubs for control or data transfer.

The final transformation of the existing code into the Microdriver approach requires the programmer to complement the driver code which can be performed using user-level debugging and instrumentation aids. In fact, the costs of the Microdrivers architecture are the burden on programmers to convert existing drivers to Microdriver’s, in the form of annotating driver and kernel code.

Over the years, the Microdriver architecture was further extended. Security mechanism were introduced in Microdrivers architecture mediating and checking the communication between the u-driver and the corresponding k-driver [72]. In this model, the authors introduced a technique to automatically infer data structure integrity constraints to be enforced by the Remote Procedure Call (RCP). A u-driver communicates with the corresponding k-driver through RPC. When the k-driver receives a request from the kernel to execute functionality implemented in the u-driver, such as initializing or configuring the device, it forwards this request to the u-driver. Similarly, the u-driver may also invoke the k-driver to perform privileged operations or to invoke functions that are implemented in the kernel. However, the u-driver is untrusted and all requests that it sends to the k-driver must be monitored. The RPC monitor ensures that each message conforms to a security policy and checks both data values and function call targets in these messages. The RPC monitor also ensures that the k-driver function calls that are invoked by the u-driver are allowed by a control transfer policy that is extracted using static analysis of the driver.

Decaf drivers [73] further extends the Microdrivers architecture to allow existing Linux kernel drivers to be incrementally converted to Java programs in user mode. The aim is to improve driver reliability through simplifying driver development and allowing most driver code to be written in user level languages, to take advantage of the language's type and memory protections.

2.6 Virtual Machines

Virtualization is the simulation of a hardware platform, storage devices and network resources. It has been a subject of research for more than forty years [113]. Nowadays, where computers are sufficiently powerful, virtualization can be used to present the illusion of running several operating systems instances in one single machine. IBM VM/370 [109] was one of the first systems to use virtualization to support the execution of legacy code.

Platform virtualization is performed on a given hardware by a Virtual Machine Monitor (VMM) or Hypervisor, which creates a simulated computer environment – the Virtual Machine (VM). In a virtualized environment, it is desirable to run DDs inside the VM, rather than in the VMM, for reasons of error containment and reduction in the software engineering effort. By running the drivers in a VM, a bug in the driver does not compromise the VMM or the others VM. It also avoids the re-

implementation of the entire driver infrastructure in the VMM, and instead, there is simply a reuse of the driver support already present in the guest operating system.

One of the major sources of performance degradation in virtual machines is the cost of virtualizing I/O devices to allow multiple guest VMs to securely share them. While the techniques used for virtualizing CPU and memory present near native performance [110][111][112], it is challenging to efficiently virtualize most I/O devices. Each interaction between a guest OS and an I/O device needs to undergo a costly interception and validation by the virtualization layer (VMM and VM) to ensure isolation, data multiplexing and demultiplexing.

Xen [110] is an x86 VMM that can run many instances of different operating systems in parallel on a single physical machine (host). The XEN VMM runs immediately after the bootloader during the machine start-up. It executes directly on the host hardware and is responsible for handling CPU, memory and interrupts.

Supervised by the VMM, XEN runs several instances of domains (VM) totally isolated from the hardware, which means that they have no privilege to access the existing devices or I/O functionalities. However, Domain 0 is a specialized VM with special privileges to directly access the hardware, handling access to the system's I/O functions and interaction with other VM. The Domain 0 kernel contains the drivers for all the devices in the system and also has a set of control applications to manage the creation, destruction and configuration of VM. Figure 2-1 depicts the XEN architecture.

Xen supports two virtualization techniques: i) hardware assisted virtualization and ii) paravirtualization. The first approach resorts to extensions recently introduced in the machines, namely the Intel VT or AMD-V hardware extensions. In this mode, XEN uses Qemu [168] to emulate the PC hardware, including the BIOS, IDE disk controller, VGA graphic adapter, and other devices. This technique does not require any change on the OS that runs in the VM. However, due to the full emulation overhead, virtualized VM are usually slower.

Paravirtualization is a virtualization technique that presents a software interface to virtual machines that is similar, but not identical to that of the underlying hardware. The intention is to reduce the portion of the guest's execution time spent performing operations that are substantially more difficult to run in a virtual environment compared to a non-virtualized environment. The paravirtualization provides specially defined routines to allow the guest and host to request and acknowledge these tasks, which would otherwise be executed in the virtual domain degrading performance. A paravirtualized platform may allow the VMM to be simpler, shifting

the execution of critical tasks from the virtual domain to the host domain, and/or reduce the overall performance degradation of machine-execution inside the virtual-guest.

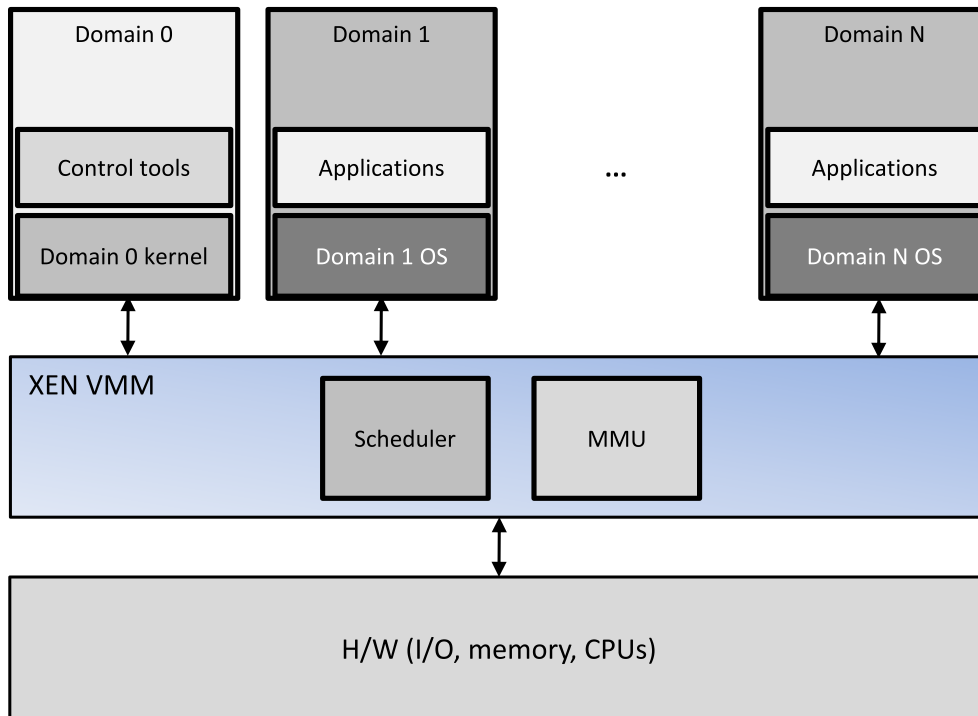


Figure 2-1: Xen architecture.

Paravirtualization requires a XEN paravirtualized-enabled kernel and paravirtualized drivers so that the VM is aware of the VMM and can run efficiently without virtual emulation of the hardware. The same is to say that changes need to be performed to the OS running on paravirtualized VM. XenoLinux was the first paravirtualized enabled kernel.

The Xen VMM uses an I/O architecture that is similar to the hosted VMM architecture [114]. As depicted in Figure 2-2, it employs privileged domains, called Driver Domains, which uses a Linux native DD to access I/O devices directly, and perform I/O operations on behalf of other unprivileged domains, called Guest Domain. The guest domains resort to virtual I/O devices controlled by paravirtualized drivers to request the driver domain for access to devices.

To virtualize network access, Xen provides each Guest Domain with a number of virtual network interfaces, which the Guest Domain uses for all its network communication. Each virtual interface in the Guest Domain is connected to a corresponding backend network interface in the Driver Domain, which in turn is

connected to the physical network interfaces trough bridging, IP routing or NAT based solutions.

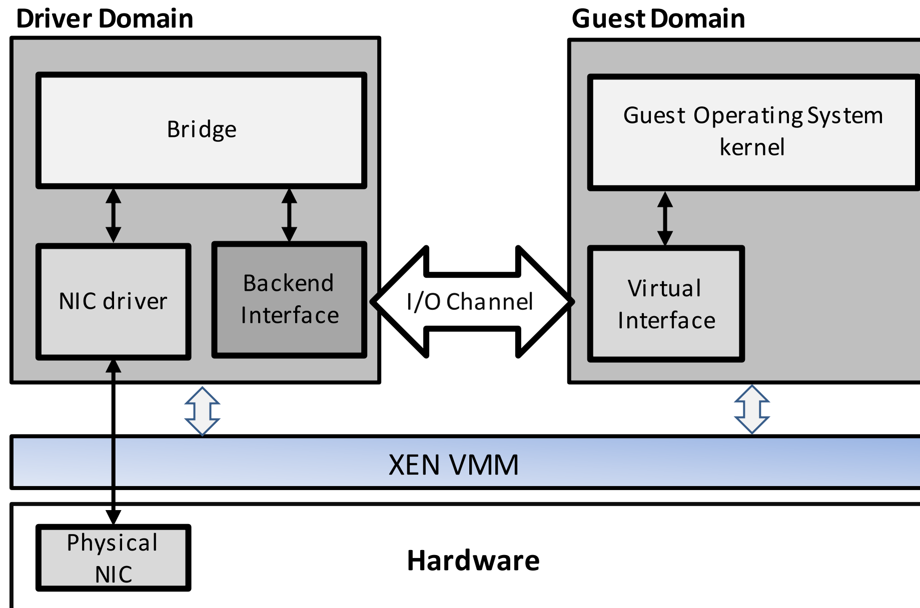


Figure 2-2: Xen network driver organization.

Data transfer between the virtual and backed network interface is achieved over an “I/O channel”, which uses a zero-copy page remap mechanism to implement the data transfer. The combination of page remapping over the I/O channel and packet transfer over the bridging provides a communication path for multiplexing and demultiplexing packets between the physical interface and the guest’s virtual interface.

Several research works showed that Driver Domains run with poor performance [74][75] and therefore, there were alternatives proposals for XEN aiming to improve efficiency. For example, TwinDrivers [75] semi-automatically partitions DDs into a performance-critical part and a non-performance-critical part. In this approach, Xen runs the performance-critical part of the DD inside the VMM and the no-performance-critical part in the Driver domain.

Despite the advantages of a virtualized system a fault in a VMM’s DD can affect all other VMs. Although the VMM is relatively reliable because it is developed and published by a closed group, and subject to a lot of tests, the DD codes, used either by the VMM or privileged VM are mostly unreliable, since most DDs are developed independently by other groups.

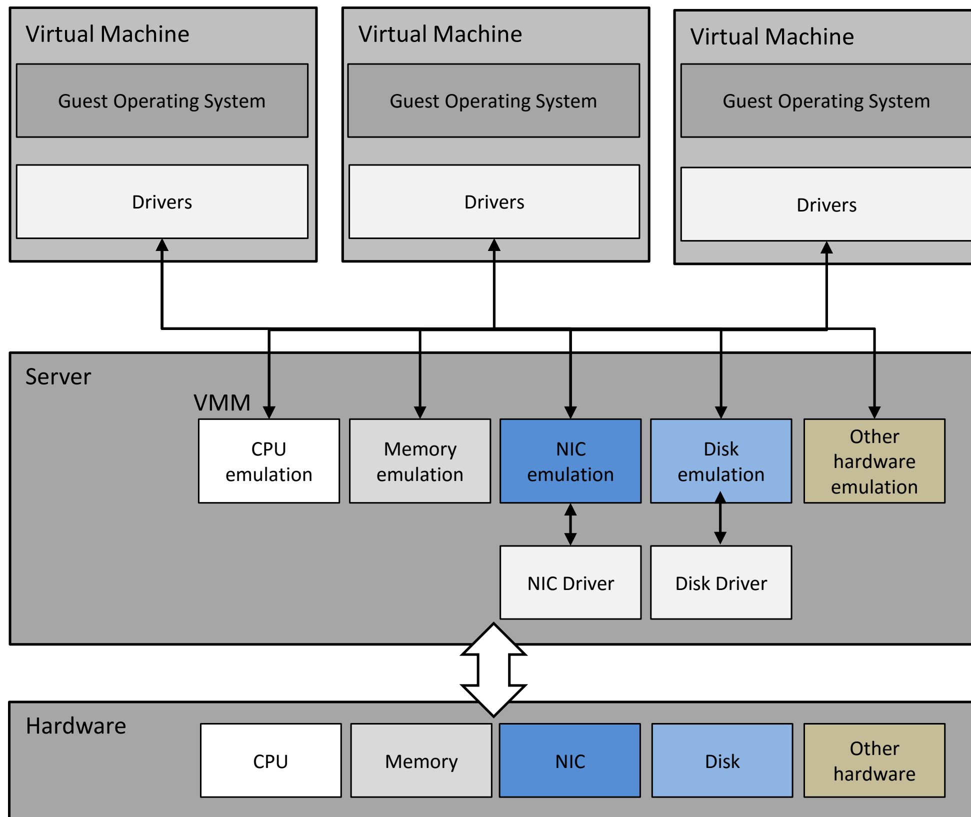


Figure 2-3: VMware architecture.

VMware [111][112] is one of the most popular software platform that allows multiple virtual machines to share hardware resources on a single hardware. The execution schedule and the sharing of resources give the illusion that each VM is running directly on a dedicated hardware platform. Unlike Xen, where the VMM relies on a separate operating system in the Domain 0, VMWare was designed specifically for virtualization with no need for another operating system.

The architecture of VMware is depicted in Figure 2-3. The implementation for I/O was designed taking into consideration the need to handle performance critical devices such as the network and disk. In this architecture, an I/O request issued by a guest OS is first handled by the driver of the guest operating system in the VM. Since the VMware emulates specific hardware controllers, the corresponding drivers will be loaded in the guest VM.

The privileged IN and OUT instructions used by the virtual devices to request I/O accesses are trapped by the VMM and handled by the device emulation code based on the specific I/O ports being accessed. The VMM then calls the device independent network or disk code to process the I/O request.

This approach, allows for unmodified operating systems to run in each VM:

- The drivers used by the guest operating system will most likely be always the same, since the architecture always presents the same emulated devices. Since these drivers will be used by most, if not all implementation solutions, a bug present in these drivers can potentially affect all such systems.
- The drivers used by the VMM are designed targeting specific hardware and maintaining the same upper interface and these drivers need to be developed and maintained by VMware.

2.7 Summary

The Windows OS defines the WDM which provides a unified approach for all kernel-mode DD. It consists in a layered driver architecture where every device is serviced by a driver stack. The WDM specifies an architecture and design procedures for several types of devices which implies a well-defined structure from which the DD and the OS can interact.

Windows DDs expose functions that provide services to the OS. However, only one function is directly known by the OS, as it is the only one that is retrieved from the binary file when the driver is loaded. Other interface functions are registered by the DD in the OS to service specific purposes depending on the DD type.

The executable file of a DD follows the same format used to represent applications and dynamic link libraries. The executable file contains the binary code, relocation information and dependencies of the DD from other software.

In Linux, a DD is named a “module” and consists in a piece of code that can be added to the kernel at runtime. Linux modules follows a unified device model for the kernel and contains abstractions that feature out commonalities from DDs. Similarly to the Windows OS, the binary transport file of a Linux module is known and can be interpreted to identify the sections of the binary machine code, relocation information and dependencies.

In monolithic OS, DD share the same privileges as the remaining kernel components. Therefore, an error in a DD (or module) can compromise the dependability of a system. Microkernels were developed with the idea that OS functionality, such as DDs, protocol stacks and file systems, would be implemented as a user-space program, allowing them to be executed like any other process which could minimize the consequence of errors in this type of software. Mach [104][105], Chorus [106] and MINIX3 [69][70] are examples of microkernel systems to take this approach.

In the Microdrivers architecture [71], a DD is split into a kernel-level k-driver and a user-level u-driver. The critical path code, such as I/O, and high-priority functions, such as interrupt handling, are implemented in the k-driver. This code enjoys the full speed of a pure kernel driver. The remaining code, which is invoked infrequently, is implemented in the u-driver and executes outside the kernel in a user-mode process. When necessary, the k-driver may invoke the u-driver. Microdrivers is an approach to isolate DD execution and minimize the effects of bugs in this type of software in the dependability of the entire system.

Virtualization is the simulation of several system components, including the hardware platform, storage devices and network resources. Platform virtualization is performed on a given hardware by a Virtual Machine Monitor (VMM) which creates a simulated computer environment – the Virtual Machine (VM). In a virtualized environment, DDs run inside the VM, rather than in the VMM, for reasons of error containment and reduction in the software engineering effort. By running the drivers in a VM, a bug in the driver does not compromise the VMM or the others VM. Despite the advantages of a virtualized system a fault in a VMM's DD can affect all other VMs. Although the VMM is relatively reliable because it is developed and published by a closed group, and subject to a lot of tests, the DD codes, used either by the VMM or privileged VM are mostly unreliable, since most DDs are developed independently by other groups. Xen [110] is an x86 VMM that can run many instances of different operating systems in parallel on a single physical machine (host).

VMware [111][112] is a popular software platform that allows multiple virtual machines to share hardware resources on a single hardware. Unlike Xen, where the VMM relies on a separate OS in the Domain 0, VMWare was designed specifically for virtualization with no need for another operating system. Since the VMware emulates specific hardware controllers, the corresponding drivers will be loaded in the guest VM. The privileged IN and OUT instructions used by the virtual devices to

request I/O accesses are trapped by the VMM and handled by the device emulation code based on the specific I/O ports being accessed. The VMM then calls the device independent network or disk code to process the I/O request.

The approach taken by VMware allows for unmodified operating systems to run in each VM. The drivers used by the guest OS will most likely be always the same, since the architecture always presents the same emulated devices. However, since these drivers will be used by most, if not all implementation solutions, a bug present in these drivers can potentially affect all such systems. Additionally, the drivers used by the VMM are designed targeting specific hardware and maintaining the same upper interface which requires these drivers to be developed and maintained by VMware.

CHAPTER 3 RELATED WORK

The detection of vulnerabilities in DDs is related to several different research areas. This chapter starts with a revision of some preparatory concepts, namely the ones related to failures and vulnerabilities of systems. More detailed descriptions about the basic concepts and taxonomy of dependable and secure computing can be obtained in [6], and other definitions can be found in [7][8].

Then, the chapter gives an overview of fault injection techniques. We will see how fault injection has been used as a methodology to evaluate systems dependability at various development stages and a few selected works will be briefly explained. We conclude this section with a description of the components that compose a generic fault injection system.

The study on robustness testing is dedicated to the explanation of several systems used in the measurement of how well a system operates when subjected to the presence of exceptional inputs or a stressful environment.

We also talk about instrumentation and dynamic analysis. This is relevant for understanding what kind of techniques can be employed to interact with DDs at function level as well as how to control their execution.

The section related to DD execution isolation complements the study about DD execution control, a trend that has been followed to increase the dependability of systems by protecting them from malfunctioning drivers.

Static analysis addresses an approach for the verification of the reliability properties of drivers by analysing the code without executing it while, at the same time, inferring misbehaviours along the program's control flow.

When addressing driver programming we overview some of the proposed changes on DD construction as an approach to eliminate the root causes that lead to driver failures, instead of dealing with their consequences.

3.1 Preparatory Concepts

A system is an entity that interacts with other entities, including, hardware, software, humans, and the physical world. These other entities are the environment of the given system. The common interface between the system and its environment is the system boundary.

The function of a system is what the system is intended to do and is described, in terms of functionality and performance, by its functional specification. The behaviour of a system is what the system does to implement its function and is described by a sequence of states. The state of a given system is composed by: computation, communication, stored information, interconnection, and physical condition. The external state is the part of the system that is perceived at the system interface. The remaining is its internal state. The sequence of the system's external states, as it is perceived by the users, is the service delivered by that system. A system can assume a role as a provider of services to other components and can assume a role as a client that expects services from system providers. A system can sequentially or simultaneously act as provider and client to other systems. The structure of a system is the set of elements that is composed of and connected in a specific manner. Thus, the system behaviour is a result of each component individual behaviour combined according to its structure. This recursive decomposition stops when an atomic element is found, i.e., an element that cannot be decomposed in other systems or its composition can be ignored.

According to the Federal Standard 1037C, a fault is an accidental condition that causes a functional unit to fail to perform its required function, a system defect [8]. In most cases, a fault causes an error in the internal state of a component and, eventually may affect the external state. A fault is active when it causes an error, otherwise it is dormant.

Faults can be characterized as a vector definition of several dimensions, including:

- System boundary: internal or external to the system
  Faults can be originated inside or outside the system's boundary and can result in errors propagated into the system by interaction or interface.

- Dimension: hardware or software
  Hardware faults are related with the physical structure of the system, electronic components and power. Software faults are related to programs and logical conditions.

- Persistence: transitory, periodic or permanent
  Faults can be transitory and therefore happen only when certain conditions are meet and with a certain degree of probability. They may also be periodic, and thus possible to forecast, or permanent and usually easier to detect.

- Level: degree of the manifestation
  Fault levels are dependent on the fault dimension. For hardware faults, the fault level measures the degree of a physical manifestation or characterization of the system, such as tension, current, power, radiation or environmental conditions. In software, the fault level measures the value of a parameter or a return value, and depends on the parameter or return type definition (integer, long, other value type).

A fault experiment is the insertion of one fault in the system under test (SUT) and the registration of its behaviour and impact. A fault injection campaign is the set of fault experiments used to exercise the SUT in order to achieve statistical confidence in the analysis of its behaviour. The faultload is the set of faults that are used in the fault injection campaign and the workload is the set of tasks that the SUT has to perform during the experiment.

An error is a discrepancy between the intended system behaviour and its actual behaviour. Errors occur at runtime when the system enters some undesired system state due to the activation of a fault.

A failure is the temporary or permanent termination of the ability of an entity to perform its required function. Failures happen because of hardware or software problems. Hardware failures are originated by physical phenomena, provoked by a faulty component that does not work like it should. Software failures are provoked by errors in the program code or data.

Modern societies depend on computers for communications, banking systems, social care, healthcare, and so many other different areas. The Dependability is

defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance [9] as:

"[...] the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [...]"

Dependability is therefore a fundamental attribute, and it can be characterized by several properties, such as, availability – the readiness for correct service, reliability – continuity of correct service, and safety – absence of consequences to the users and the environment, Integrity – Absence of improper system alteration and Maintainability – Ability to undergo modifications and repair.

Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions [37].

The term vulnerability, in computer security, can be explained as a weakness that allows an attacker to compromise a system. In order to occur a security failure, it is necessary a conjunction of a vulnerability fault and an attack that exploits this weakness, leading to an error in the system. From this viewpoint, a vulnerability represents a reduction of the system dependability attributes.

To exploit a vulnerability an attacker needs to have at least one tool and/or technique that allow him to explore the vulnerability in the system. This may be achieved either by gaining local physical access to the system or having a remote link.

Vulnerabilities are usually expressed by a product vendor as a defect requiring a patch, upgrade or a configuration change. This is the type of information that attackers search to profile an attack. Once a vulnerability is discovered, it is only a matter of time before an attacker develops a tool (worm, virus, file, information packet, etc.) that can take advantage of the fault. Exploits created to take advantage of these security vulnerabilities can lead to system compromise, non-availability, data loss, exposure of confidential information, and other losses.

Vulnerability management is the process in which vulnerabilities in information technology are identified and the risks of these vulnerabilities are evaluated. This evaluation leads to correcting the vulnerabilities and removing the risk or a formal risk acceptance by the management of an organization (e.g., in case the impact of an attack would be low or the cost of correction does not outweigh possible damages to the organization). The term vulnerability management is often confused with vulnerability scanning. Despite the fact both are related, there is an important difference between the two. Vulnerability scanning consists of using a computer

program to identify vulnerabilities in networks, computer infrastructure or applications. Vulnerability management is the process surrounding vulnerability scanning, also considering other aspects such as risk acceptance, remediation, etc.

3.2 Fault Injection

This section briefly reviews some of the work related with fault injection. A few of the techniques and ideas introduced in this area were used in our research to uncover DDs' vulnerabilities.

Fault injection is the deliberate introduction of faults into a SUT to experimentally validate its dependability. It is an important experimental technique that helps researchers and system designers to study the behaviour of the system in the presence of faults without having to wait for them to occur (which can take a long time because they are typically infrequent). This approach can be applied during all phases of the development process, including design, prototype and production phases.

To take an experimental approach, it is essential to understand the system's architecture, structure, and behaviour, including the incorporated mechanisms for fault detection and recovery. In what concerns to fault injection, the target system may be classified in one of the following major types: i) Axiomatic models, ii) Empirical processing models and iii) Physical systems.

Axiomatic models are used to describe the structure, dependability and performance of the system behaviour in the form of reliability block diagrams, fault trees, Markov graphs [148] or stochastic nets. Fault Trees (FT) are one of the most used models for reliability analysis because they represent a high-level abstraction of the system and can be solved using Binary Decision Diagram techniques. However static FT cannot handle sequential and functional dependencies between components. To overcome this lack of modelling power, a number of dynamic methodologies have been developed and used for dependability analysis of dynamic systems based on Markov Chains (MC) formalisms. On the other hand, as systems being built are increasingly complex and large, they are becoming more difficult to model and analyse. Manually generating an MC describing the system's behaviour is a daunting and an error prone task. Furthermore, MC are faced with state space explosion problem where the states to be generated grows exponentially with the number of components comprised in the system.

Empirical processing models incorporate complex or detailed behavioural and structural descriptions that require a simulation approach to process them. When systems are at the conceptual and design stages, simulation-based fault injection

tools can be used to evaluate the dependability of the system. In some cases, this is the most suitable approach because it enables the testing of the system using cost-effective tools. The results from the evaluation allow the design team to review their implementation options, without having to compromise the project, as it might happen at a more advanced phase. At this point, the system is a series of abstractions and assumptions where most of the implementation details are still to be defined.

Physical systems can correspond to prototypes or final systems that are implemented in hardware and/or software. In this case, systems can be distinguished as being composed of hardware-only, software-only, hardware and software. The results obtained with simulation-based fault injection tools are often biased by some design assumptions, since there are usually differences between the model and the final implementation. When evaluating a prototype or final setup system this effect usually disappears because the SUT corresponds to an instance of the final version. This is important since the actual workload can impact on the performance of the error handling mechanisms as pointed out by several studies [10][11].

Simulation-Based Fault Injection

Computational models of systems and their implementation in simulation software are used in testing during the early design phases, without the expense of developing a prototype. They may present different levels of abstraction, such as, device level, functional block-level, protocol level or system level. The level of detail of the model influences the accuracy of its behaviour. Highly detailed models may take too much time to simulate due to the size of the system's activity. On the other hand, lighter models may be faster to run but may not accurately represent the systems mechanisms due to the implemented abstractions.

Simulation-based fault injection can be performed in simulators of the computational models of the systems [13][14]. Here, the injection software may act at different levels of abstraction, modifying the structural organization of the SUT, the communication links between components, or the component models.

One representative solution that implemented simulation-based fault injection is DEPEND [12]. It is an integrated simulation environment for the design and dependability analysis of fault-tolerant systems. It provides facilities to rapidly model a fault-tolerant architecture and conduct extensive fault injection studies. DEPEND is a functional process-based simulation tool, where the system behaviour is described by a collection of processes that interact with one another. To develop

and execute a model, the user writes a control program in C++, using the objects available from the DEPEND library that simulate the hardware components (e.g., CPUs, communication channels and disks). The fault-tolerant characteristics of an object, the type and the method by which faults are injected are also specified by the user. The program is then compiled and linked with the DEPEND objects, and the resulting model is executed in the simulated run-time environment. Here, the assortment of objects, including the fault injectors, CPUs and communication links, run simultaneously to simulate the functional behaviour of the complete system. Faults are injected, according to the user's specification, and a report containing the essential statistics of the simulation is produced. The results are then used by the development team to perform the necessary corrections in the developing project.

Hardware Emulation Based Fault Injection

Simulation-based fault injection can be too time consuming as the system models become excessively complex and detailed. Field Programmable Gate Array (FPGA) circuits allow the implementation, with a high degree of accuracy, of emulation models of hardware components. Since hardware circuit design normally uses some kind of Hardware Description Language (HDL), the FPGA can be programmed to mimic the intended hardware. This opens a window of opportunity for the execution of fault injection experiments into system models within a reasonable time and having most of the advantages of simulation-based fault injection.

One of the first approaches based on FPGA emulation systems employed the concept of Dynamic Fault Injection (DFI) [15]. A typical use of the FPGA involves the following steps: 1) Provide an HDL of the circuit to implement; 2) Generate the connection list (netlist) with all the connections defined; 3) Transfer the netlist to the FPGA and 4) Use the FPGA. To perform a fault experiment using an FPGA, the above sequence of steps must be performed reconfiguring the HDL to include the fault. DFI explores the possibility of reducing the number of FPGA reconfigurations in a fault campaign by previously identifying which faults are dependent of using extra hardware. The extra hardware is connected to input ports of the FPGA and acts as demultiplexer activating, one at a time, each dependent fault.

Hardware-Implemented Fault Injection

Hardware-implemented fault injection refers to the process of injecting faults in a physical system. In most cases, a processor was chosen as the target because the system behaviour is mainly determined by this component. In addition to this

argument, the following reasons also justify the interest of the fault injection at the processor pins:

- Faults injected in the processor pins can reproduce not only internal processor faults but also memory and bus faults, and most of the faults in peripheral devices. For instance, faults in a peripheral device can be duplicated by injecting faults in the processor pins during the cycles in which the processor is reading data from the peripheral device;
- It is possible to cause errors in other parts of the target system by injecting faults in the processor pins. For example, a fault injected in the processor data bus, during a memory write cycle, will cause an error to be stored in the addressed memory cell.

Hardware implemented fault injection comprehends several techniques, among them pin-level fault injection [16][17][18][19], test access port fault injection [23][24][25], electro-magnetic interference fault injection [21][22] and radiation based fault-injection [26][149][150]. We will briefly describe each of these techniques in the following sections.

Pin-level Fault Injection

Pin-level fault injection is one of the most common methods of hardware implemented fault injection [16][17][18][19][20][153]. Here the injector probe has physical contact with the target Integrated Circuit (IC) and directly interferes with the electric signals of the system. Since the faults are created at the pin level, they are not identical to traditional faults that occur inside the IC. Nevertheless, many of the same effects can be observed.

The change of the electrical currents and voltages at the IC pins can be achieved through two main techniques: i) Active probes and ii) Socket insertion. Active probes add an electrical current to the circuit attaching the probe to the pins of the IC, without removing the chip from the system board. This method can provide stuck-at faults (maintaining a certain current level in the pin) or bridging faults by placing a probe across two or more pins of the IC.

With socket insertion, a socket is placed between the target hardware and the system board. The contact between the IC pins and the circuit board, provided by the socket, is controlled by the fault injector. This technique extends the faults that can be performed, supporting the insertion of signals that are the result of a logic operation involving previous signals of the pin itself or any other pin. The main

advantage of the socket insertion technique over the active probes is the level of isolation that can be achieved relative to the surrounding circuitry.

MESSALINE is an example of a tool for physical pin-level fault injection [16]. It has the ability of creating faults at the IC pin level such as: i) IC pins disconnected from the system board; ii) IC pins connected to a specific electric voltage level; iii) IC pins are connected together and iv) other complex logical signal combination as the result of a logic combination of other electric signals. It is a composition of four modules. The Fault Injection Module enables the generation of faults at the IC pins. The Activation Module uses physical output interfaces to initialize and control the target system. The ReadOut Module is responsible for reading the values present on selected target IC pins as a result of the experiments and finally, the Software Management Module creates the test sequence, does the run time control of its execution and collects the results to be used in the post-test analysis.

Test Access Ports Fault Injection

The miniaturization of device packaging, the development of surface-mounted packaging, and the associated development of the multi-layer board reduced the physical access for insertion of probes. The advances in semiconductor industry required software and hardware tools that could access critical functionalities of the IC. The standards IEEE-ISTO 5001-2003 (Nexus) [23], IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture (JTAG) [24] and the proprietary Background Debug Mode (BDM) [25] provide solutions to interface VLSI circuits (microprocessors and FPGA) equipped with built-in debugging and testing features. They define I/O Test Access Ports (TAP) that enables the observation of the IC internal state, registers and other elements. Furthermore, TAP allows the injection of faults into the pins and internal state elements of the IC. The type of faults that can be injected depends on the debugging and testing features supported by the target IC.

A fault injection experiment through TAP involves: a) defining a breakpoint and then wait for the program to reach the breakpoint, b) read the value of the target location, c) manipulate the value, d) write the new faulty value back to the target location, and e) resume the program execution. The main advantage of TAP fault injection is that faults can be inserted internally in IC without making any changes to the system's hardware or software. Examples of the Test Access Port fault injection tools and applications can found in [151][152].

Electro-magnetic Interference Fault Injection

Electro-magnetic interference (EMI) is produced by a wide range of sources, such as, motor cars, trains and industrial plants. Since computer systems are in environments were such sources exist, electro-magnetic interference fault injection has also been applied in various scenarios.

Typically, the EMI tests are conducted inside of an anechoic chamber with a controlled RF environment where the SUT is placed (e.g., between two metal plates which in turn are connected to the EMI generator). The isolation provided by the anechoic chamber provides assurance that the observed results are resultant from the EMI tests and not from an external source. The source of the EMI then combines different signal power with focus in a particular signal frequency, systematically swiping a large spectrum of inject random frequencies.

The impact of EMI is usually much more severe than the impact of other commonly used injection techniques. Since EMI and in particular Power Supply Disturbances tend to affect many bits, which can modify a larger part of the system state [21][22][153].

Radiation-Based Fault Injection

As the dimensions and operating voltages of electronics are reduced, their sensitiveness to radiation increases dramatically. There is a multitude of radiation effects in semiconductor devices that vary in magnitude from data disruptions to permanent damage. This is a primary concern for commercial terrestrial and space applications.

Radiation based fault injection is a contactless hardware implemented fault injection. Here the injector does not have direct contact with the target system, but produces some physical phenomenon that potentially influences the behaviour of the target electronics (e.g., by generating some sort of radiation).

Fault injection by heavy-ion radiation is a technique for creating faults in systems, especially inside the ICs [26][149][153]. This method however is difficult to apply to existing computers mainly because the target chip outputs have to be compared pin-by-pin with a gold unit, in order to know whether the radiation has produced errors inside the target IC or not. Since the heavy-ions are attenuated by molecules and other materials in the irradiation path, the target circuit must be run in a vacuum. Consequently, the packaging material that covers the target chip must be removed. This is a major difficulty because commercial IC components are many times destroyed during the removal of the packaging material.

A major feature of the heavy-ion fault injection technique is that faults can be introduced into VLSI circuits at locations impossible to reach by other methods, such as pin-level fault injection. The faults are also reasonably well spread within a circuit, as there are many sensitive memory elements in most VLSI circuits. Thereby, the injected faults generate a variety of error patterns which allow a thorough testing of fault handling mechanisms.

Software-Implemented Fault Injection

Computer systems are nowadays too complex for the mechanisms associated with fault activation and error propagation to be completely understood. This makes the evaluation of dependability properties a very demanding task. Analytical modelling becomes extremely hard and only possible if a great number of simplifying assumptions is used. Although hardware fault injection evaluation is suitable to validate specific fault handling mechanisms, the design of specialized tools is almost impossible as their complexity is directly associated to the control and check of the fault effects of the system being evaluated.

Software Implemented Fault Injection (SWIFI) is primarily motivated to avoid the difficulties and cost inherent to physical fault injection approaches and is intended to emulate both software and hardware faults. Compared to hardware fault injection tools, it has lower complexity and development effort, as there is no need to build specialized hardware. A SWIFI tool also presents a greater degree of portability, since it can be applied to several different systems with little modifications.

SWIFI tools can emulate hardware faults using mainly two different approaches applied to the software: i) at compile-time and ii) during runtime. In the compile-time approach the injector modifies the target program source code to insert some errors, which causes faults to be activated when the code is executed. The modified code potentially alters the functional behaviour of the original program, while it emulates the effect of hardware or software faults.

To inject faults at runtime one must use a mechanism that suspends the workload in the SUT, calls the injector code and resumes the execution of the SUT's software in the point where it was stopped. This can be accomplished by using one of the following mechanisms: a) timeout; b) exception-trap or c) code insertion. The timeout mechanism is the simplest and corresponds to the occurrence of an event triggered by a software or hardware timer that was set to expire at a certain instant. In response to the event, a routine is called to produce the fault. In the exception-trap mechanism the control of the system is transferred to the injector by means of a software trap or hardware exception. The handler routine is then responsible for the

generation of the fault. In the code insertion mechanism, unlike the compile-time code modification, the binary code of the target program is modified directly in memory at runtime. This can be accomplished, for instance, by placing the injector code in the handler routines of some advanced debugging features of modern CPUs. The handler routine is then triggered, for example, whenever the CPU's program counter reaches some predefined value.

These techniques can be used to target applications and the OS. In case of an application, the fault injector is inserted into the application itself or layered between the application and the operating system. If the target is the OS, the fault injector must be embedded in it, as it is very difficult to add a layer between the machine hardware and the OS.

However, the SWIFI approach can have some limitations: It cannot inject faults into locations that are inaccessible to software, e.g., peripheral devices; The software instrumentation may disturb the workload running on the target system and even change the structure of original software; and it usually has a poor time-resolution making this approach unable to capture certain error behaviours associated with low latency faults. This, however, can be minimized with careful design of the injection environment or by adopting a hybrid software/hardware solution (described later). The SWIFI approach can also have fidelity problems due to poor time-resolution. For long latency faults, such as memory faults, the low time-resolution may not be a problem. For short latency faults, such as bus and CPU faults, the approach may fail to capture certain error behaviour, including some forms of error propagation.

This problem can be solved by taking a hybrid approach, which combines the versatility of software fault injection and the accuracy of hardware monitoring [35]. The hybrid approach is well suited for measuring extremely short latencies. However, the hardware monitoring involved can have high costs and decrease flexibility, by limiting observation points and data storage size.

Over the years, several tools have been proposed for SWIFI, such as FERRARI [28], FIAT [27], FINE [29], DEFINE [30] (an evolution of FINE), DOCTOR [31], FTAPE [32], GOOFI [34] and Xception [33]. As an example, we will describe in more detail the Xception toll.

Xception is a fault injection and monitoring environment that introduces faults by software and monitors their impact on the target system behaviour. This tool was fundamentally designed to emulate hardware transient faults in functional units of the target processor. It uses the advanced debugging and performance analysis features that exist in most modern processors, such as performance counters and

breakpoint registers. The counter register can be programmed to record a number of user defined events such as load, store, or floating point instructions. The breakpoint register enables the programmer to specify where to break the program for a wide range of situations such as load, store or fetch of data from a specified address or even some instruction types (e.g., floating point instructions). Using a combination of these mechanisms, faults can be injected when the instruction in a specific address is fetched or when the data stored in some address is accessed. In practice, the exception trigger that inserts the faults is programmed in the processor debugging hardware before starting the target application. This allows the target application to be left unchanged and be executed at normal speed (and not in some special trace mode). When trigger is reached, the trigger handler creates the fault. Since Xception operates at the exception handler level, and not through any service provided by the operating system, the injected faults can affect any process running on the target system including the OS.

Components of a Fault Injection System

Figure 3-1 represents the most relevant components usually employed in Fault Injection Systems (FIS). These components implement activities such as disturb the execution of the SUT, observe the behaviour and determine if the fault was tolerated.

The FIS actions are defined by the System Controller that is in charge of coordinating the experiment and of synchronizing all other components. It can run in the SUT itself or in a separate machine.

In each round of the fault injection campaign, the Setup Module prepares the system to become operational and meet the desired initial conditions.

The Workload Generator stimulates the SUT to perform its tasks. This component is used to exercise the system at normal or stressful conditions, depending on how fast it demands/provides services at the SUT interface.

Faults are produced by the Fault Generator that either creates them at runtime, commanded by the Controller, or prior to the experiment. In this later case, the faults are stored in a Fault Library for later use. The Fault Injector injects the faults into the SUT, either interacting physically and/or logically with it.

The Data Collection is in charge of capturing, processing and analysing the data produced in the SUT. The collected information is saved in a Data Storage.

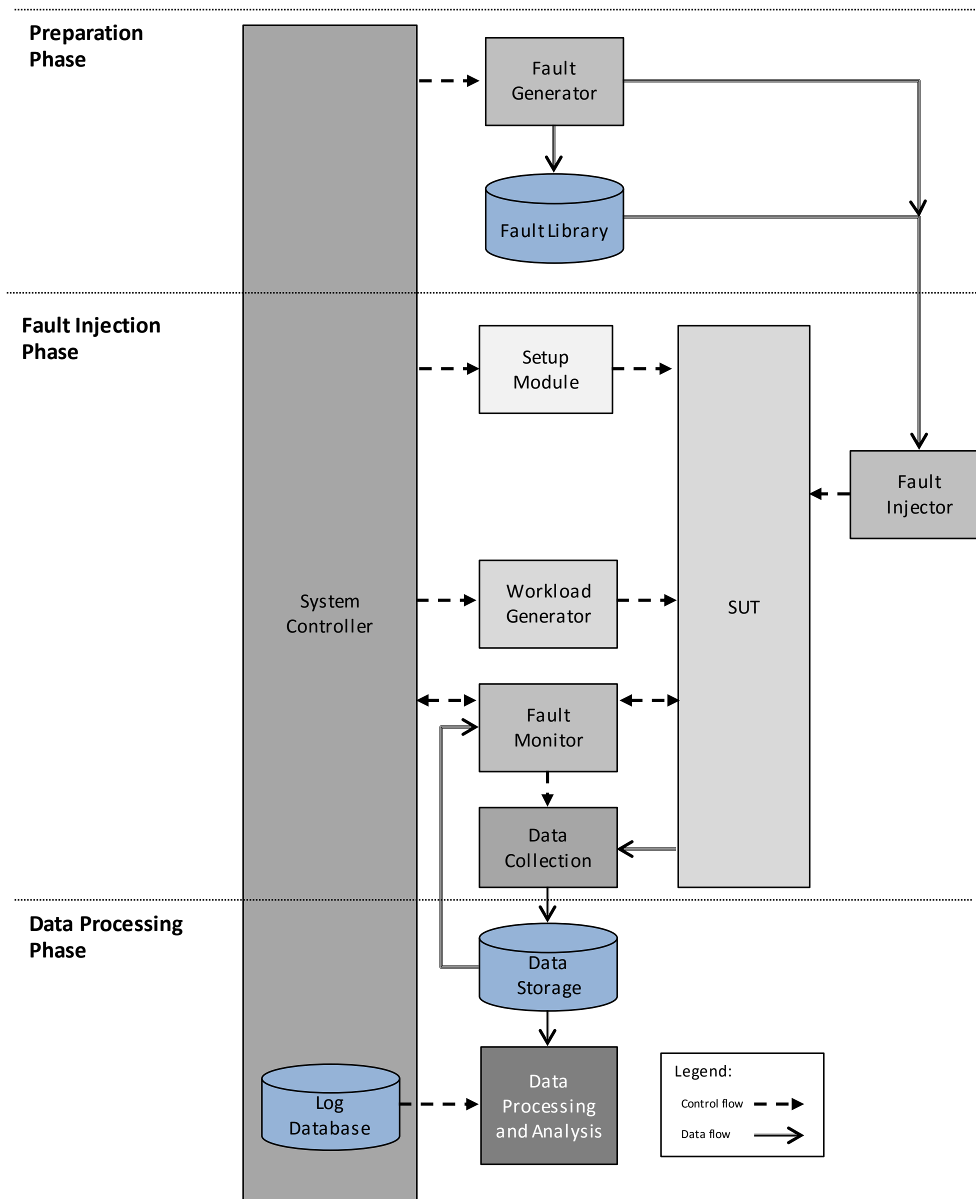


Figure 3-1: Basic components of a fault injection system.

The Fault Monitor observes the SUT behaviour and gives feedback to the System Controller, allowing it to decide the next round in the injection campaign.

The Data Processing and Analysis component allows the analysis of the SUT behaviour even after the fault campaign has finished. All FIS activity is registered in a Log Database for complementary analysis (e.g., sequence of events).

Typically, there are 3 different phases in the activity of the FIS:

1. Preparation: The preparation phase is the stage where all the preliminary conditions are setup. In the cases were the faults can be previously generated it also involves the creation or loading of the Fault Library;

2. Fault Injection: The fault injection phase represents the stage where the fault injection campaigns occur; The system is setup, including the workload, faults are injected and the behaviour of the system is observed.
3. Data Processing: In this phase, the processing and analysis of the results of the tests is performed.

3.3 Robustness Testing

In computer science, robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. Robustness testing is an experimental evaluation technique which forces incorrect inputs and/or stressful situations to systems or system components, trying to activate faults that result in incorrect operation.

The acceptability of robustness testing is based on the ability to reproduce the initial conditions, the observations and measurements of the experiments. Thus, an important aspect of the robustness testing is the establishment of result metrics, which form the basis for evaluation and comparison. For instance, the 5-point CRASH [36] scale organizes the failures caused by the injection of faults, according to the severity of their effect on an end system, being ‘C’ (catastrophic) the most severe and ‘H’ (hindering) the less one. Others failure modes scales have also been proposed, for instance [3][53][54][55].

Robustness testing has been employed in some proposals for dependability benchmarking approaches [154][155]. One of the main targets of robustness testing has been the OS interfaces, which have been tested with erroneous inputs being inserted at the application interface (see for instance [38][43][46]). By creating evaluation techniques that provide a direct, repeatable, quantitative assessment of OS exception handling abilities, developers may obtain feedback, for instance, about the capability of a new OS version to protect itself. Knowledge about the exception handling weak spots of an OS enables system designers to take extra precautions by increasing the type of validations they perform on input/outputs. Additionally, quantitative assessment enables system designers to make comparison whether it might be more robust to use a COTS OS than an existing proprietary OS. For instance, some studies have shown that open source solutions did not exhibit significantly more critical failure modes than commercial ones [45][52]. Other studies on dependability benchmarking include CORBA middleware implementations [67][68] and Online Transaction Processing (OLTP) systems [39].

Robustness testing can also be used to evaluate the dependability of systems at the DD level. It can be employed for instance to verify how well the DD software can cope with erroneous inputs. During the course of our research we have explored some of the ideas related to this area to build a system capable of measuring the robustness of the OS when subject to DD malfunction. In the rest of this section, we review in more detail several robustness testing tools and describe some of the experimental results that were obtained. A particular focus will be given to tools that address the robustness of DDs.

Operating System Robustness Testing

FUZZ was an early attempt to perform OS robustness testing, and it targeted the system utility applications [40]. FUZZ was capable of producing random printable and control characters, which were then used as input to the utility applications. Automatic testing was achieved by utilizing a script that initiated the applications and passed the random data.

This tool was used to test a large collection of utilities running on several versions of the Unix OS (and was later applied to other OS) [47]. Three types of failure modes were considered in the test campaigns:

1. Crash - the program ended abnormally producing a core file;
2. Hang - the program appeared to loop indefinitely, or
3. Succeed - the program terminated normally.

The first results showed a surprising number of programs that would crash or hang. Pointer/array errors, unchecked return codes, input functions, were pointed as some of the root causes of the observed behaviour. Although the problems affected a large number of regularly used OS utilities, many of the discovered problems were still present in new OS versions several years later [41]. In the recent years, other researchers have extended these ideas into more intelligent and less random tools, capable of testing different kinds of software components (see for example [48][49][50]).

An example of an early method for automatically testing OS for robustness is the CRASHME tool [42]. It operates by writing random data values to memory, and then it spawns a large number of tasks that attempt to execute those random bytes as concurrent programs. While many tasks terminate almost immediately due to illegal instruction exceptions, on occasion a single task or a confluence of multiple tasks can cause an operating system to fail. If run long enough CRASHME may eventually get lucky and find some way to crash the system.

The Random and Intelligent Data Design Library Environment (RIDDLE) was used to stress testing the Windows NT software [51]. RIDDLE generates input for the application being tested using the grammar of the component under analysis, rather than simply creating random input. It can combine for instance random field values with boundary value conditions to evaluate a program behaviour under anomalous conditions. RIDDLE was employed to compare the reliability of native Windows NT utilities with the Cygnus Win32 port of the widely-distributed GNU utilities. The results show that the native Windows NT utilities had far fewer failures due to anomalous input than the GNU Win32 utilities, which revealed that errors may arise when porting a stable program from one platform to another.

Another example of a software robustness testing system tool that automatically tests the exception handling capabilities of the OS is BALLISTA [43]. While it can be used for testing APIs beyond OS (e.g., a simulation framework), much of the focus of the evaluation was on POSIX OS interfaces. BALLISTA testing methodology involves automatically generating sets of exceptional parameter values that are used as arguments when calling software modules. The results of these calls are examined to determine whether the software module detected and notified the calling program of an error, or whether the task (or even the system) suffered a crash or hang as the result of a call.

The evaluation of Microkernels fault handling mechanisms was the target of the Microkernel Assessment by Fault Injection Analysis and Design Aid (MAFALDA) [44]. MAFALDA takes advantage of the debugging features of most modern microprocessors to inject faults by software and monitor their effects (as in Xception [33]). One form of fault injection implemented by MAFALDA consists in the corruption of the input parameters. It simulates the propagation of an error from the application level to executive level of the microkernel, aiming to evaluate the robustness properties of the microkernel interface. It traps the target kernel so that an exception is automatically raised whenever there is a call to an entry point of the microkernel. The handler for this exception is responsible for the corruption of the input parameters and once injected the handler lets the call proceed to the kernel.

Device Driver Robustness Testing

Device Path Exerciser (DPE) is a tool for testing the reliability and security of drivers [59]. It calls drivers through a variety of user-mode I/O interfaces with valid, invalid, and poorly-formatted data that will cause some error in the driver execution if not managed correctly. These tests can reveal improper driver design or implementation that might result in system crashes or might make the system vulnerable to malicious

attacks. During a test, DPE sends an enormous quantity of calls (hundreds of thousands) to the driver in rapid succession. The calls include changes in data access methods, valid and invalid buffer lengths and addresses, and permutations of the function parameters that might be misinterpreted by a flawed parsing or error-handling routine. The tool verifies that calls sent to the driver are completed correctly and do not cause system crashes, system memory pool corruption, or memory leaks. The driver is expected to handle each of the requests properly, either by returning valid data or by rejecting the request.

IoSpy and IoAttack are tools that perform IOCTL and Windows Management Interface (WMI) tests on kernel-mode drivers [60]. These tools help to ensure that the drivers' IOCTL and WMI code validate data buffers and buffer lengths correctly, avoiding buffer overruns that can lead to system instability. When a device is enabled for testing, IoSpy captures the IOCTL and WMI requests sent to the driver of the device, and records the attributes of these requests within a data file. IoAttack then reads the attributes from this data file, and uses these attributes to fuzz, or randomly change the IOCTL or WMI requests in various ways, before sending them to the driver. This allows further entry into the driver's buffer validation code without writing IOCTL or WMI-specific tests.

Plug and Play (PnP) related code paths in the driver and user-mode components can have their robustness evaluated by the Plug and Play Driver Test Tool [61]. This tool forces a driver to handle almost all the PnP IRP, and more specifically it stresses three main areas: removal, rebalance, and surprise removal. The tool provides a mechanism to test each of these separately or to test them all together. This PnP testing is accomplished by using a combination of user-mode API calls (through the test application) and kernel-mode API calls (through an upper filter driver).

3.4 Instrumentation and Dynamic Analysis

Analysing the dynamic behaviour, performance, and correctness of software and systems is invaluable to software developers and hardware designers. Instrumentation is done by inserting debugging and profiling information. It supports monitoring and measurement of the level of performance of the application and writes execution traces to the display or files to help the diagnose of errors. In fact, the ability to interfere with systems and software is the building block for software fault injection and robustness testing presented in the previous sections. Having access to appropriate source code, it is often trivial to insert new instrumentation or extensions by rebuilding the applications or the OS to provide necessary insights about its execution. When no source code is available, the ability to instrument

unmodified binaries facilitates the analysis of commercial applications in realistic scenarios.

Next, we will briefly introduce some of the existing tools that address instrumentation and dynamic analysis. We are especially interested in understanding what techniques and challenges are involved in instrumenting DDs.

Detours [90] is a library for intercepting arbitrary Win32 binary functions on x86 machines. The interception code is applied dynamically at runtime by replacing the first few instructions of the target function with an unconditional jump to a user-provided detour function. The removed instructions from the target function are preserved in a trampoline function, which also has an unconditional branch to the remainder of the target function. The detour function can either completely replace the target function or extend its semantics by invoking the target function as a subroutine through the trampoline. Detours experiments were based on Windows applications and DLLs, but were not applied to DDs.

A software system that performs run-time binary instrumentation of Windows applications is PIN [91]. PIN collects data by running the applications in a process-level virtual machine. It intercepts the process execution at the beginning and injects a runtime agent that is similar to a dynamic binary translator. To use PIN, a developer writes a “Pintool” application in C++ using the PIN API consisting of instrumentation, analysis and call-back routines. The “Pintool” describes where to insert instrumentation and what it should do. Instrumentation routines walk over the instructions of an application and insert calls to analysis routines. Analysis routines are called when the program executes an instrumented instruction, collecting data about the instruction or analysing its behaviour. Call-backs are invoked when an event occurs, such as a program exit. Several applications were instrumented using PIN, such as Excel and Illustrator. PIN executes in user level ring3, and therefore can only capture user-level code. Another example of a dynamic binary translation technique similar to the one used by PIN is implemented by DynamoRio [103].

NTrace [102] is a dynamic tracing tool for the Windows kernel capable of tracing system calls, including the ones involving drivers. The used technique is based on code modification and injection of branch instructions to jump to tracing functions. It relies on the properties introduced by the Microsoft Hot patching infrastructure, which by definition start with a `mov edi, edi` instruction. NTrace replaces this instruction with a two-byte jump instruction. However, due to the space constraints, the jump cannot direct control into the instrumentation routine. It rather redirects to the padding area preceding the function. The padding area is used as a trampoline into the instrumentation proxy routine.

DDT [81] combines virtualization with a specialized form of symbolic execution to test DDs. This tool uses a modified QEMU [168] machine emulator together with a modified version of the Klee symbolic execution engine [147]. DDT runs a complete, unmodified, binary software stack, comprising of the Windows OS, the drivers to be tested, and all associated applications. DDT forces the loading of the driver of interest, determines the driver's entry points, coerces the OS into invoking them, and then symbolically executes the DD of interest using an adapted version of Klee.

3.5 Isolation of Device Driver Execution

Commodity operating systems are built using a monolithically design where all the operating system functions run in kernel mode. To simplify the design of the kernel, components such as DDs, dispatcher and file systems share the same address space without isolation. However, with this unconstrained access, every bug in these components can potential compromise the system correctness.

The isolation of the kernel from other operating system components could increase system dependability. Once the kernel has been well tested, a flaw in any other component, especially the ones that change often (such as DDs), could no longer compromise the entire system. Furthermore, the kernel could integrate recovery procedures to restore the faulty component by restarting the service, eventually with minor or no losses in data or context.

CPU manufacturers have incorporated in their architectures hierarchical domains to protect data from functionality faults. Unfortunately, mainstream operating systems do not take fully advantage of these features. In this section, we are going to give an overview of the mechanisms aiming to isolate DDs. We will focus our attention in techniques involved in the protection of the system from DD failures, used in testing a DD without corrupting the entire system, and employed to record DD faults whenever they occur.

Runtime Protection

Software fault isolation (SFI) [78] is a software technique used to isolate the execution of individual applications and prevent faults from these applications to contaminate the remaining system. In this technique, the untrusted software is placed inside a fault domain consisting on a contiguous region of memory within an address space. The virtual address space of the untrusted software is divided into

aligned segments such that all virtual addresses within a segment share the same segment identifier.

Two mechanisms were proposed to enforce the execution of the code within its fault domain: i) segment matching and ii) address sandboxing. In the first mechanism, the binary of the application to be isolated is modified to include some checking code before every unsafe instruction. If the checking code determines that the target address is safe, it lets the application to proceed. Otherwise, the inserted code traps to a system error routine outside the distrusted module's fault domain. The second mechanism employs address sandboxing, which consists in inserting some code before each unsafe instruction to set the value of the segment identifier, forcing it to stay inside the same fault domain. Although it cannot catch the illegal addresses, it prevents the untrusted code to affect any other domain. The prototype used in SFI targeted user applications, but the proposed ideas could also be applicable to isolate DDs into SFI segments.

In an alternative approach, the OS could be divided into inner kernel and application resources, as suggested by VINO [115]. The inner kernel cannot be modified by applications but processes can override the behaviour of the application resources. Files, directories, threads, transactions, physical memory pages, virtual memory pages and queues are example of resources each one includes properties and default operations implementation. New resource types are added to VINO by compiling them it into the kernel.

VINO, uses a trusted compiler that generates code with either bounds checking or sandboxing to ensure code safety [82]. The generated code is digitally signed so that all code installed in the kernel can be verified to be from the trusted source. The compiler also ensures that the generated code does not mask interrupts or modifies itself. Each graft receives its own heap and stack, and when a graft changes kernel state (e.g., by opening a file), the kernel records the fact so that any such modifications can be undone if the graft misbehaves. If the process is aborted, the corresponding transaction is aborted, and the system is returned to a consistent state.

Static Verification and Runtime Memory Protection

XFI [83] is a protection mechanism designed for Windows running on the x86 hardware platform that combines static verification with run-time software guards for memory access control and system state integrity.

The XFI-rewriter produces XFI binary modules from Windows x86 executable (EXE, DLL or SYS). It makes use of debug information (PDB files), to distinguish

code from data and to add structured guards and verification hints to be used later during the loading process.

Guards consists on code added to the binary modules intent to enforce that an XFI module complies with the policies that dictate interaction with its system environment: memory access constraints, control flow (the code can never flow outside the module's code, except via calls to a set of prescribed support routines, and via returns to external call-sites); stack integrity; authorized instruction execution; system-environment integrity (e.g., segment registers cannot be modified).

In addition to restricting interactions between a module and its host, XFI places constraints on the execution of the module through: control-flow integrity (execution follow a static, expected control-flow graph); program-data integrity (Certain module-global and function-local variables can be accessed only via static references from the proper instructions in the module); Assured self-authentication (a module authenticates itself to the host system).

The correctness of XFI protection depends on the load time verification of the XFI module. XFI-verifier makes a linear pass over the bytes of an XFI module checking statically that each XFI module has the appropriate structure and the necessary guards. Verification also considers the execution of machine-code instructions abstractly; it manipulates verification states which are predicates that describe concrete execution states. A trusted XFI module requires that it passes all verifications of a defined policy and that those policies hold during its execution. It can be seen as an example of proof-carrying code (PCC) [116], even though they do not include logical proofs.

LXFI [86] isolates faults in a DD by checking its accesses to kernel API, according to programmer-specified integrity rules. LXFI uses a compiler plug-in to instrument the generated code to grant, check, and transfer capabilities between kernel modules.

The main goal of LXFI is to prevent an adversary from exploiting vulnerabilities in kernel modules in a way that leads to a privilege escalation attack. LXFI protection relies in the control of the functions that a module is allowed to call, in the verification of its control flow and in the data structure integrity used by the module. The application of LXFI is a four step process where: i) developers annotate core kernel interfaces to enforce API integrity between the core kernel and modules, ii) module developers annotate certain parts of their module where they need to switch privileges between different module instances; iii) LXFI's compile time rewriter instruments the generated code to perform API integrity checks at runtime and iv)

LXFI's runtime is invoked at the instrumented points, and performs checks to uphold API integrity (if the checks fail, the kernel panics).

Low Level Driver Execution Isolation

One reason that explains many failures in commodity operating systems, is the close integration between untrusted extensions and the core kernel, which violates the principle of least authority. In particular, since new DDs are often introduced in the system, it is difficult to ensure that all of them behave correctly. Therefore, some proposals have suggested the use of low level isolation mechanisms to prevent failures in the drivers from propagating to the rest of the system. Some examples of these solutions are presented next.

One of the first approaches to provide isolation of DDs on a commodity operating system was Nooks [76]. It seeks to achieve: i) DD execution isolation, ii) automatic recovery of the DD with iii) minimum changes to existing systems.

The isolation performed by Nooks is achieved by memory management to implement lightweight protection domains with virtual memory protection, and the Extension Procedure Call (XPC), to transfer the control safely between DDs and the kernel.

The memory management ensures that the kernel has read-write access to the entire memory space while DD is restricted to read-only access. The XPC mechanism provides a function to pass control from the kernel to the DD and another to pass control from the DD to the kernel. These transfer routines save the caller's context on the stack, find a stack for the calling domain (which may be newly allocated or reused when calls are nested), change page tables to the target domain, and then call the function. The reverse operations are performed when the call returns.

Nooks interposes on extension/kernel control transfers with wrapper stubs to perform the following tasks: i) check parameters for validity by verifying with the object tracker and memory manager that pointers are valid; ii) implement call-by-value-result semantics for XPC, by creating a copy of kernel objects on the local heap or stack within the extension's protection domain and iii) perform a XPC into the kernel or extension to execute the desired function.

Nooks recovers from a DD failure by recording all resources that are held and when a failure is detected, the isolation components releases the resources and then tries to restart the driver.

Herder et al. [79] suggest a way to isolate DDs by enforcing least authority and refining the driver by extensive software-implemented-fault-injection testing. These

principles, intent to limit the damage that can result from accidents or errors. It also reduces the number of potential interactions among privileged programs so that unintentional, unwanted, or improper uses of privilege are less likely to occur.

Another example of a fault isolation technique is proposed in Byte Granularity Isolation (BGI) [77]. BGI is implemented as a compiler plug-in that generates instrumented code for DDs and links it to an interposition library that mediates the communication between the DDs and the kernel.

BGI runs DDs in controlled memory regions (domains) separated from the kernel and trusted DDs. It associates an Access Control List (ACL) with each byte of the virtual memory to the domains that can access it and how they can access it. Access rights are granted and revoked by code inserted by BGI compiler and by the interposition library according to the semantics of the operation being invoked. The protection is enforced by inline checks inserted by BGI and by checks performed by the interposition library.

The interposition library contains kernel wrappers that are called by the DD and DD function wrappers that are called by the kernel.

The kernel wrapper checks the rights to the arguments supplied by the DD, can revoke the rights to some of those arguments, it calls the wrapped kernel function, and it may grant rights to some objects returned by the function. The DD function wrapper may grant rights to some arguments, it calls the wrapped DD function, it may revoke rights to some arguments, and it checks values returned by the DD.

This way BGI can grant access to the bytes that a domain should access and it can check accesses to these bytes regardless where they are in memory. Additionally, it controls when a domain is allowed to access these bytes because it grants and revokes the access to the specified bytes.

BGI required modifications to the kernel to reserve virtual address space for the kernel table when the system boots, and to reserve virtual address space in every process when process are created to create the domains.

3.6 Static Analysis

For long the development of applications has been made easier because of compilers that are able to identify program errors related to syntax, type violations, and mismatches between a function’s formal and actual parameters. More sophisticated checking includes looking at pointers and uninitialized variables. However, most of the analysis is done intra-procedurally, and consequently problems caused by the interactions between functions are not detected. Additionally, these techniques are not applicable to many categories of defects, such

as memory leaks, buffer overflows, resource consumption and `NULL` pointer assignments to name a few. Another form of more sophisticated testing must therefore be applied to increase the quality and reliability of the software.

Static analysis techniques analyse a program without executing it, but follow all paths while building an internal representation of the program's control flow. Over the years many tools have appeared, and one way to classify them is based on the type of flaws that are searched for as the ones enumerated by the Common Weakness Enumeration (CWE) classes [131] or the Seven Pernicious Kingdoms taxonomy [132].

- Input validation and representation: Input validation and representation problems are caused by metacharacters, alternate encodings and numeric representations. Security problems result from trusting input.
- API abuse: An API is a contract between a caller and a callee. The most common forms of API abuse are caused by the caller failing to honour its end of this contract. For example, if a program fails to call a correct sequence of functions.
- Security features: Incorrect handling of security features in topics such as authentication, access control, confidentiality, cryptography, and privilege management.
- Time and state: Defects related to unexpected interactions between threads, processes, time, and information, deadlocks, race conditions.
- Errors: Errors related with error handling.
- Code quality: Poor code quality of the code leading to unpredictable behaviours, especially under system stress.
- Encapsulation: Poor software boundaries leading to data leakage between users and debug code leftovers.
- Environment: Everything that is outside of the source code but is still critical to the security of the product that is being created.

The Input Validation and Representation category looks into bugs that are caused by meta characters, alternate encodings and numeric representations, and security problems resulting from trusting input. Examples of bugs in this category are buffer overflows, command injection, cross-site scripting, format string, integer overflow, SQL injection, etc. This category includes several of the bugs normally

reported as security vulnerabilities by tool vendors. Tools that support both timing and state, and input validation and representation bugs include:

- Coverity [133][134][146], a C, C++ and Java checker;
- Jlint [135][136], a checker of Java class files that is based on data flow and abstract interpretation;
- PREfast [56], a C, C++ checker based on intra-procedural analysis and statistics;
- Splint [137], a C lint prototype for security vulnerability analysis based on taint annotations;
- Archer [138], a C array checker that uses symbolic analysis;
- FindBugs [139], a Java checker that uses bug-patterns and data flow analysis on Javaclass-files;
- Gramma Tech's CodeSonar [140], a C,C++ checker that performs whole-program, inter procedural analysis.

The Timing and State category looks into bugs that are due to distributed computation via the sharing of state across time. Examples of bugs in this category are dead locks and race conditions. Tools that support this category of bugs include:

- JPF [95][128], a Java programming language checker that model-checks annotated Java code;
- PREfix [141], a C/C++ checker based on inter-procedural data flow analysis;
- ESP [142], a C checker that focuses on scalability of analysis and simulation;
- Goanna [143], a C/C++ checker that model-checks static properties of a program.

The Security Features category is concerned with authentication, access control, confidentiality, cryptography and privilege management. Examples of bugs in this category are insecure randomness, least privilege violation, missing access control, password management and privacy violation. A tool that supports timing and state, input validation, and security features is Veracode [144], a binary/executable code checker based on data flow analysis that performs penetration testing on the binary code.

The API Abuse category is concerned with the violation of the (API) contract between a caller and a callee. Examples of bugs in this category are dangerous

functions that cannot be used safely, directory restrictions, heap inspection, and various often misused language or operating system features. Tools that support timing and state, as well as API abuse bugs include:

- SLAM [58][144], a C/C++ DD checker that model-checks and verifies code against a specification of a DD;
- A tool that support timing and state, input validation, security features and API abuse bugs is Static Code Analysis [14].
- Other uses of static analysis approaches have been applied to the detection of viruses and worms [93][123][124][125]. Also, it has been applied to the detection of rootkits [126] and spyware-like behaviour [127].

The research on static analysis tools is by far exhaustive. However, from the sample, it can be apprehended that most of the static analysis tools requires access to the source code or annotations. Binary static analysis tools also exist but they face additional challenges since need to deal with machine code representation which difficult the analysis. In this kind of tools typically a pre-processing phase is performed to translate the binary code to its internal representations as is the case of [94] (see also RevGen [97] and LLVM [96] representation).

Our interest in these tools is quite clear, static analysis tools can play a role in the detection of vulnerabilities and errors of DDs. However, many of the static analysis tools require changes in the source code to be effective, which is something not easy to get for commercial operating systems. There are however a few techniques that can operate over binary code.

We will describe some of the existing work related with static analysis and understand how we may benefit from static analysis to help us find vulnerabilities in DDs.

Compile time static analysis

PREfast is a static verification tool that examines each function of the driver code independently, for the detection of general syntax and coding errors, such as unchecked return values [55]. The driver-specific features detect subtler errors, such as leaving uninitialized fields in a copied I/O Request Packet (IRP) and failing to restore a changed Interrupt Request Level (IRQL) by the end of a routine.

PREfast has to know additional information about the source code in the form of annotations. These annotations are special macros that are expanded into meaningful definitions only when PREfast runs. General-purpose and driver-specific annotations are defined in header files that must be included in the code. The

annotations extend function prototype and describe the contract between the function and its caller. This enables for PREfast to analyse the code more accurately, with significantly fewer false positives and false negatives. Annotations also make the code easier to read, forming a documentation that does not drift apart from the code.

Static Driver Verifier (SDV) is a source level compile-time tool that explores code paths in a DD by symbolically executing the source code [57][58][121]. SDV automatically creates an alternative program that is an abstraction of the original program. The alternative program is then checked against API usage rules using a state machine. The program abstraction is expressed as a Boolean program that has all the control-flow constructs of the original code (including procedures and procedure calls) but only Boolean variables. SDV uses a symbolic model checking algorithm based on binary decision diagrams [122] to determine if the Boolean program obeys to the API usage rule. SDV places a driver in a hostile environment and systematically tests all code paths looking for violations of WDM usage rules. The symbolic execution makes very few assumptions about the state of the OS or the initial state of the driver, so it can exercise situations that are difficult to analyse by traditional testing.

Runtime Checking

Static analysis has also been employed to detect implementation flaws or deficiencies in input validation or device responsiveness as is the case of Carburizer [80]. It uses CIL [130] and intermediate language and tool set for analysis and transformation of C programs to read the pre-processed C code of the driver and produce an internal representation of the code suitable for static analysis that locates dependencies on inputs from the device. When it finds in the code a control decision, such as a branch or a function call, based on data from the device, the analyser marks the data as sensitive because it is dependent on the correct functioning of the device. Similarly, if the driver code uses a value originating from a device in an address calculation, such as an array index, the use of the address is also dependent on the device and thus marked as possibly unsecure.

Carburizer inserts the necessary code to report a failure if the data is incorrect. Additional code is also generated aiming to detect stuck interrupts and non-responsive devices. In the case of problems with the device, the added code invokes a generic recovery service that can reset the device using shadow drivers [129] to provide this service.

Static verification of binary code

In [94] is described an approach for the identification of use of data coming from untrusted sources in x86 executables in ELF binary format. It performs the analysis of the binary program assembly level representation of the program. During the analysis of the program, indirect `call` and `jump` instructions are attempted to be resolved to help in the identification of functions and the derivation of a complete control flow graph. To resolve the jump-table-based branches, the code is backtracked in the code until the instruction that set up the jump table access is reached, thus recovering the base location and the number of entries in the table.

Mechanisms are applied to detect loops. Recursive function calls are identified by applying a standard topological sort algorithm on the function call graph of the program.

The resolution of the library functions used in the program to test is performed by combining the information contained in the Procedure Linkage Table and the relocation table of the binary.

The analysis technique uses symbolic execution of functions to determine a set of possible targets and approximates all possible concrete executions and focus on identifying insecure uses of the standard C library functions.

3.7 Driver Programming Model

Among the main reasons behind buggy drivers are low-level programming language, poorly-defined communication protocols between the DD and the OS, a complex driver execution infrastructure and a multithreading computational model. To address these difficulties, efforts were made to provide safer programming languages and a friendlier driver execution infrastructure.

Commodity OS such as Windows and Linux and their extensions are built using mainly the C language, which gives a high level of freedom to the programmers namely to make mistakes.

To be effective the driver programming model approach needs to be adopted by DD writers and sponsored by both OS and device manufacturers since they require changes in the current development paradigm as well as access to protected information.

Introducing changes to existing programming languages can improve the quality of driver building, with an increase of the overall dependability. However, requiring the use of totally different languages and building procedures may be a challenge.

Formal specification is a way to have a clear and well defined contract between the DD and the rest of the system. It relies on the accurate and detailed information to minimize bugs. Eventually, automatic tools can then formally analyse the resulting code and identify specification violations either statically or dynamically.

The replacement of the multi-threaded model with an event-based model can be a solution to reduce (or eliminate) some of the difficulties related with concurrency, one of the most common problems in driver development.

In this section we will describe some example works that aim to achieve correctness by construction as opposed to fault detection and isolation. The goal is to eliminate the root causes that lead to faults instead of dealing with their consequences.

Type Based Checking and Restart Capabilities

SafeDrive [85] aims to improve DD reliability by adding type-based checking and restart capabilities to existing DDs written in C language. The primary goal of SafeDrive is to detect memory and type errors ensuring that data of the correct type is used in kernel API calls and in shared data structures, preventing the kernel or devices from receiving incorrect data.

To transform a driver written in C into one that obeys stricter type safety requirements there is the need to fix the C languages constructs that can cause violations without requiring extensive rewrites. SafeDrive uses Deputy, a type system for pointers that can enforce memory safety by using annotations in header files for APIs and shared structures. The annotations express known relationships between variables and fields (e.g., `int * count(len) buf` means that the variable `len` holds the number of elements in `buf`). Programmers are responsible for inserting type annotations that describe pointer bounds expressing known relationships between variables and fields

Deputy is implemented as a source-to-source transformation that runs immediately after pre-processing. During compilation the annotations are transformed into appropriate run-time checks. At run time a SafeDrive extension is loaded into the same address space as the host system and is linked to both the host system and the SafeDrive runtime system. The SafeDrive runtime system checks the compliance with the assertions and tracks the use of resources that are being requested by the driver to the OS. If assertions fail, SafeDrive invokes the recovery subsystem that will use its internal data structures to restore the resources used by the driver.

Laddie [88] introduced a type-safe language that enables driver writers to create I/O interfaces between a driver and its device so that these I/O interfaces cannot be easily misused. A Laddie specification consists in a set of declarations that form I/O rules for reading and writing into the registers of a device. The rules are pre-conditions and post-conditions for reading and writing each register. Each specification is organized in two different sections. The first one is where the components for the logical state of the device are declared. The second part is where the I/O rules for communicating with the device are set.

To produce a DD a programmer need to go through the following stages: i) produce a Laddie specification; ii) compile the Laddie specification and produce Clay output files [120]; iii) write the body of the DD in Clay language and iv) compile Clay files to obtain the driver. During this compilation stage a series of verifications are run to ensure that all types are declared and that rules are consistent. The consistency tests will catch errors where no inputs could satisfy the conditions.

Clay's compiler will do all the compile time checking and inform the programmer if any run-time checks are still necessary to be included in the driver code.

Formal Specification

Writing formal specifications has associated challenges since they derive from the device and OS specifications and documentations itself that seldom undergoes adequate quality assurance causing the formal specification derived from such information to reproduce defects in addition to extra ones introduced during the formalisation process.

Distilling device specifications from existing driver implementations is another possible approach to construct a device specification. However, access to source code is usually not the case for commercial OS. Besides, a DD may contain errors, which may be carried over to the resulting specification. A third approach to construct a device specification is to derive it from the register transfer level (RTL) description of the device written in a hardware description language while abstracting away most of internal logic and modelling only interface modules. However, access to the RTL description is usually not viable since it is part of the device manufacturer's intellectual property.

Termite [87] is a DD synthesis tool that uses a combination of formal specifications of the device's registers and behaviour and the interface between the device and the OS to produce a less error prone working DD.

The device interface specification describes the programming model of the device, including its software-visible states and behaviours. The OS interface

specification defines the services that the driver has to provide to the rest of the system, including the services available from the OS to the driver.

Given the specifications, the Termite algorithm implements a driver in C language that satisfies two main requirements: i) safety (the driver shall not violate the specified order of operations) and ii) liveness (the driver is required to perform all its actions within a finite number of steps).

The construction of a device is performed in three steps. The first step combines individual driver interface specifications into a single specification. The second step produces a driver state machine that has safety and liveness properties. The third step translates the state machine into a driver implementation in C.

The formal specification of a DD is written in a high-level language and is therefore not as error-prone as developing the DD itself. Errors in specifications can be reduced by using model checking techniques. Thus, generating the code automatically from the formal specifications reduces programming errors in drivers since a bug in the driver can only occur as a result of an error in the specification.

Event Based Model

Currently in modern OS the driver functions are mainly called by the kernel when it needs to perform an I/O or deliver an interrupt notification to the driver. However, since kernels are multithreaded, the driver needs to be prepared to handle concurrent invocations by multiple threads. This increases complexity since the functions of the driver need to be constructed in such a way that do not deadlock the all system and have synchronization mechanisms to hold these evocations.

As an alternative to the traditional multithreading approach, Dingo proposes the use of an event driven model [84]. Dingo also provides Tingu, a formal language for describing driver software protocols for a clear and unambiguous description of requirements of driver behaviour.

In Dingo, a driver software protocol is the collection of protocols that regulates the communication between the driver and the hardware device and the OS. This communication occurs over ports, which are bidirectional message-based communication points. Each port is associated with one protocol that defines the messages that can be exchanged, constraints on ordering, time control and contents. A protocol is violated if, after entry into a state, the given amount of time passes without triggering a transition leading to a different state.

The Tingu compiler generates a protocol observer from the Tingu specification of its ports. It intercepts all messages exchanged by the driver and keeps track of the

state of all its protocols. Whenever the driver or the OS fails to comply with the messages timings and/or contents the observer notifies the OS about the failure.

While Dingo does not eliminate bugs caused by an incorrect implementation of the protocol, the presence of a clear and complete specification of the protocol tends to reduce the occurrence of these bugs.

Another example system that proposes to reduce the complexity of driver development by changing to an active event-driven model is the Active DD architecture [89]

The active DD architecture [89], similar to Dingo, deals with synchronization issues as well as provides a clear driver control flow by assigning a dedicated thread to a DD. This driver thread receives requests from the kernel via message passing in an event-based way.

3.8 Summary

Fault injection deliberately introduces faults into a SUT to experimentally validate its dependability. In what concerns to fault injection, the target system may be classified in one of the following major types: i) Axiomatic models, ii) Empirical processing models and iii) Physical systems.

Simulation-based fault injection involves computational models of systems and their implementation in simulation software. Highly detailed models may take too much time to simulate due to the size of the system's activity. On the other hand, lighter models may be faster to run but may not accurately represent the systems mechanisms due to the implemented abstractions. A representative tool of implemented simulation-based fault injection is DEPEND [12].

Field Programmable Gate Array (FPGA) circuits allow the implementation of Hardware Emulation Based Fault Injection through emulation models of hardware components. The FPGA can be programmed to mimic the intended hardware opening a window of opportunity for the execution of fault injection experiments into system models within a reasonable time and having most of the advantages of simulation-based fault injection.

Hardware-implemented fault injection refers to the process of injecting faults in a physical system. In most cases the processor was chosen as the target because the system behaviour is mainly determined by this component. Several hardware-implemented fault injection techniques were developed such as: i) Pin-level fault injection where the injector probe has physical contact with the target Integrated Circuit (IC) and directly interferes with the electric signals of the system (see for instance [16][17][18][19]); ii) Test Access Ports Fault injection uses I/O Test Access

Ports (TAP) allows the injection of faults into the pins and internal state elements of the IC [23][24][25]; iii) Electro-magnetic interference fault injection uses a wide range of sources to produce electro-magnetic interference into the systems [21][22]; and iv) Radiation-Based fault injection that uses radiation (e.g.; heavy-ion) to potentially influences the behaviour of the target electronics [26][149][150].

Software-Implemented Fault Injection (SWIFI) is primarily motivated to avoid the difficulties and cost inherent to physical fault injection approaches and is intended to emulate both software and hardware faults. It presents lower complexity and development effort than hardware fault injection tools and can emulate hardware faults with high degree of control. Some proposed SWIFI tools include FERRARI [28], FIAT [27], FINE [29], DEFINE [30], DOCTOR [31], FTAPE [32], GOOFI [34] and Xception [33].

Robustness testing is an experimental evaluation technique which forces incorrect inputs and/or stressful situations to systems or system components, trying to activate faults that result in incorrect operation. One of the main targets of robustness testing has been the OS interfaces, which have been tested with erroneous inputs being inserted at the application interface (see for instance [38] [40][42][43][44][46][51][59][60][61]).

Instrumentation is done by inserting debugging and profiling information into the system. It supports monitoring and measurement of the level of performance of the application and capture execution traces to help the diagnose of errors. Having access to appropriate source code, it is often trivial to insert new instrumentation or extensions into systems. When no source code is available, the ability to instrument unmodified binaries facilitates the analysis of commercial applications in realistic scenarios. Some example of instrumentation tools include, Detours [90], NTrace [102] and DDT [81].

Commodity operating systems are built using a monolithically design where all the operating system functions run in kernel mode. However, with this unconstrained access, every bug in these components can potential compromise the system correctness. The isolation of the execution of kernel components have been proposed using Runtime Protection [115], Static Verification and Runtime Memory Protection [83][86] and Low Level Driver Execution Isolation [76][77][79].

Static analysis techniques analyse a program without executing it. Static analysis tools follow all paths while building an internal representation of the program's control flow. Most of the static analysis tools requires access to the source code or annotations. Complex systems can take too long or being impossible to analyse. Binary static analysis tools also exist but they face additional challenges since need

to deal with machine code representation which difficult the analysis. Examples of static analysis tools were given in the following categories: i) compile time static analysis; ii) runtime checking and static verification of binary code. The following are examples of static analysis tools, Coverity [133][134][146], Jlint [135][136], PREfast [56], Splint [137] (see also [95][128][138][139][140][141][142][143]).

Among the main reasons behind buggy drivers are low-level programming language, poorly-defined communication protocols between the DD and the OS, a complex driver execution infrastructure and a multithreading computational model. Introducing changes to existing programming languages can improve the quality of driver building, with an increase of the overall dependability.

Formal specification is a way to have a clear and well defined contract between the DD and the rest of the system. It relies on the accurate and detailed information to minimize bugs. Eventually, automatic tools can then formally analyse the resulting code and identify specification violations either statically or dynamically.

The replacement of the multi-threaded model with an event-based model can be a solution to reduce (or eliminate) some of the difficulties related with concurrency, one of the most common problems in driver development.

CHAPTER 4 ROBUSTNESS TESTING OF THE WINDOWS DRIVER KIT

Device Drivers are one of the major sources for system malfunctions. Previously we have explained a series of potential causes that contribute for this situation. In the case of monolithic OS architectures, the main reason for these problems can be attributed to the fact that the driver executes with the same privileges as the OS kernel. Since it is very difficult to change the existing software architecture, researches have proposed solutions to minimize the effects of faults in the drivers either by executing the driver code in a separate environment (from the kernel) or by wrapping the code with enough controls to prevent faults from compromising the overall system execution. The success of these solutions depends on how effective are the designed mechanisms to cope with all sorts of flaws that a DD may have.

The undeniable fact is that DDs are becoming the most dynamic and larger part of the OS code and, with new devices released frequently, this problem can grow exponentially. In this chapter, we study in some detail the effect of DD faults on the dependability of a system and determine how the OS is prepared to cope with them. This study can help us identifying some common types of faults that may lead to system failure and contribute to devise solutions that could prevent them more effectively.

In this part of our investigation we focus on the interface between the DD and the OS. As we are especially interested in dealing with Windows DDs we designed a

methodology to evaluate the robustness of the DD Kit (DDK) functions. We also built a tool that implements the methodology, which has its roots in the Ballista [43] approach. In our tool several test drivers are generated, containing DDK function calls with erroneous arguments. The argument values were selected specifically for each function, and they emulate seven classes of typical programming errors.

4.1 The Test Methodology

In a robustness testing campaign one wants to understand how well a certain interface withstands erroneous input to its exported functions. Each test basically consists on calling a function with a combination of good and bad parameter values, and on observing its outcome in the system execution. As expected, these campaigns can easily become too time consuming and extremely hard to perform, especially if the interface has a large number of functions with various parameters, since this leads to a combinatory explosion on the number of tests that has to be carried out.

This kind of problem occurs with the Windows DDK because it exports more than a thousand functions. However, from the group of all available functions, some of them are more commonly used than others, and therefore these functions potentially have more impact in the system. Moreover, in most cases, (good) parameter values are often restricted to a small subset of the supported values of a given type.

Based on these observations, we developed a methodology to test the Windows DDK. It has several steps that are implemented by a set of tools, as represented in Figure 4-1. The DevInspector tool performs an automatic analysis of the target system to obtain a list of available DDs. Then, it measures the presence of each imported function from the DDK by each driver.

Using this data, one can select a group of functions for testing, the candidate list. A XML file is manually written to describe the prototype of each function, which also includes the fault load (e.g., the bad values that should be tried).

Next, the DevBuilder tool takes as input the information contained in the XML file, a template of a DD code, some compilation definitions, and generates the workload utilized to exercise the target system and to observe its behaviour. The workload includes for each function test a distinct DD that injects the faulty input.

Other approaches could have been employed to implement the tests (e.g., a single DD injects all faulty data). However, the selected solution was chosen because: i) the control logic of each driver and management tool becomes quite simple; ii) the interference between experiments basically disappears because an OS reboot is performed after a driver test, iii) last, one can determine if the DD loading and unloading mechanisms are damaged by the injected faults.

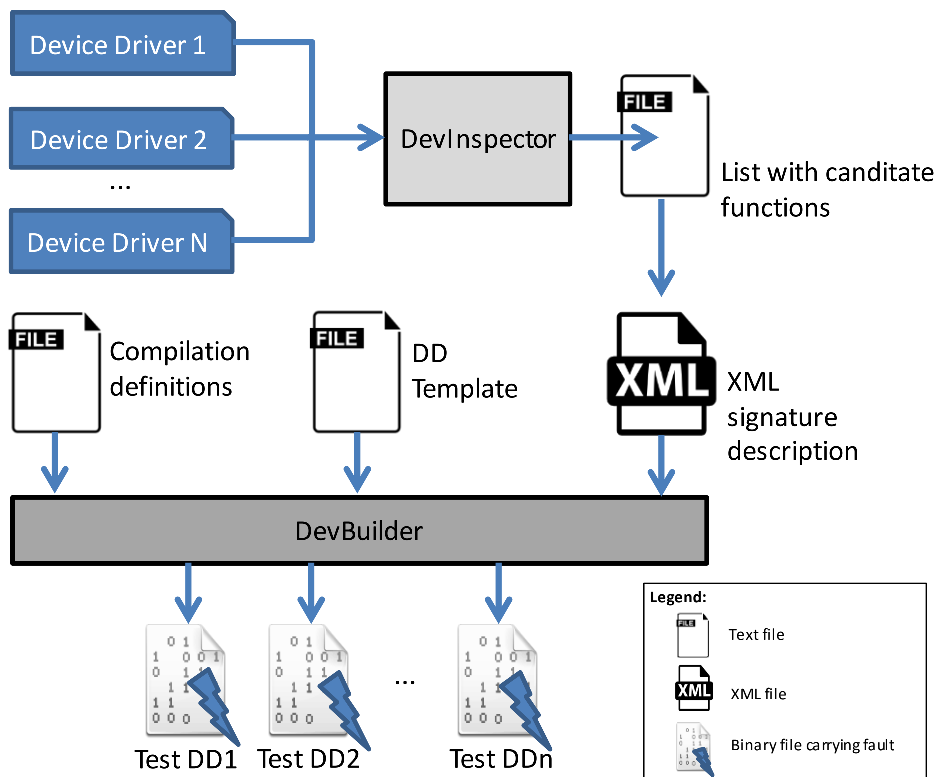


Figure 4-1: Test DD generation.

The study has looked in a comparative basis at aspects such as error containment, influence of the file system type, and the diagnosis capabilities of minidump files.

4.2 Selecting the Candidate List

Windows stores drivers in the portable executable file format [62], which contains a table with the functions that are exported from the driver and imported from the OS. In the case of drivers, the imported functions are the ones provided by the DDK. Therefore, one can discover the DD currently available in a system by looking for `.sys` files placed in `\system32\drivers`. Then, by examining the table of imported functions of the existing drivers, one can collect statistics about which DDK functions are utilized in practice.

In our experiments, we have performed several installations of Windows XP and Windows Server 2003 to use FAT32 and NTFS file systems. Windows Vista was installed only with NTFS file system. These OS and file system combinations were installed in a DELL Optiplex 170L computer. Table 4-1 shows the number of drivers found for each of our Windows installations.

Each line in the table identifies the OS name and file system, the number of drivers that were found in the OS installation and that were running when the boot sequence completed, and the number of functions imported by these drivers. As it is possible to observe, Windows Vista imports many more functions than Windows Server 2003 for roughly the same number of drivers (2400 instead of 1463).

Table 4-1: Drivers in a Windows OS installation.

OS	File System	Drivers		# of different functions in running Drivers
		Total	Running	
Windows XP	FAT32	259	93	1490
	NTFS	260	94	1494
Server 2003	FAT32	189	93	1463
	NTFS	189	92	1463
Vista	NTFS	250	113	2400

From the analysis of these drivers (both total and running), it was possible to conclude that a small group of functions was commonly present in the majority of the DD, and that most of the rest of the functions were infrequently utilized (e.g., around 900 functions were only called by 1 or 2 drivers). These results indicate that if one of the most common imported functions unsafely treats its parameters, then almost every DD is potentially affected.

For this work, the functions that were chosen for the candidate list were the ones commonly imported by the majority of the drivers. Being impossible to test every function in a reasonable time, it was used the following selection criterion:

“The tested functions had to be present in at the least 95% of all running drivers”.

Table 4-2 displays the first group of the most used functions that satisfied this criterion. In each line, the table presents our internal identifier, the name of the function and its alias (to reduce the size of the rest of the tables). We have found out that this list changes very little when this criterion is applied to all existing drivers and not only the running ones.

Table 4-3 displays the driver coverage by this group of functions in each OS configuration.

Table 4-2: Top 20 called DDK functions.

ID	Name	Alias
1	ntoskrnl::RtlInitUnicodeString	InitStr
2	ntoskrnl::ExAllocatePoolWithTag	AllocPool
3	Ntoskrnl::KeBugCheckEx	BugCheck
4	ntoskrnl::IofCompleteRequest	CompReq
5	Ntoskrnl::IoCreateDevice	CreateDev
6	Ntoskrnl::IoDeleteDevice	DeleteDev
7	ntoskrnl::KeInitializeEvent	InitEvt
8	ntoskrnl::KeWaitForSingleObject	WaitObj
9	ntoskrnl::ZwClose	ZwClose
10	ntoskrnl::IofCallDriver	CallDrv
11	ntoskrnl::ExFreePoolWithTag	FreePool
12	ntoskrnl::KeSetEvent	SetEvt
13	ntoskrnl::KeInitializeSpinLock	InitLock
14	HAL::KfAcquireSpinLock	AcqLock
15	HAL::KfReleaseSpinLock	RelLock
16	ntoskrnl::ObfDereferenceObject	DerefObj
17	ntoskrnl::ZwOpenKey	OpenKey
18	ntoskrnl::ZwQueryValueKey	QryKey
19	IoAttachDeviceToStack	AttachDev
20	ntoskrnl::memset	Memset

Table 4-3: Top 20 functions driver coverage.

OS	File System	Driver Coverage
Windows XP	FAT32	96,7%
	NTFS	96,8%
Server 2003	FAT32	96,7%
	NTFS	96,7%
Vista	NTFS	97,3%

Other selection criteria were considered, such as the static or dynamic frequency of function calls. Static frequency picks functions that appear many times in the code without taking into account the logic under it – a function may appear repeatedly in the code but may never be executed.

Dynamic frequency chooses the functions that are called most often during the execution of a given workload. Therefore, if the workload has a high file activity then disk drivers would run more, and their functions would be selected for the candidate list. This will bias the analysis towards the elected workload, which is something we decided to avoid in these experiments.

4.3 Tested Faulty Values

The main responsibility of the DevBuilder tool is to write DD based on the template code, each one carrying out a distinct function test (see Figure 4-1). To accomplish this task, all relevant data about the functions is provided in a XML signature file, and a DD source code template with special marks that identify where to place the information translated from XML into source code.

The signature file includes the function name, parameter type and values that should be tried out as well as the expected return values. In addition, for certain functions, it also contains some setup code that is inserted before the function call, to ensure that all necessary initializations are performed. Similarly, some other code can also be included, which is placed after the function call, for instance to evaluate if some parameter had its value correctly changed or to check the returned value of the performed call.

In order to obtain the relevant data about the functions, we had to resort to the Windows DDK documentation. From the point of view of a DD developer, this documentation corresponds to the specification of the DDK functions. Therefore, if there are errors in the documentation, then they may be translated into bugs in the drivers' implementations (and also in our tests). Nevertheless, in the worst case, if a problem is observed with a test, at least it indicates that the function description contains some mistake.

The signature file defines seven types of correct and faulty inputs. These values, summarized in Table 4-4, emulate the outcomes of some of the most common programming bugs.

Table 4-4: Fault type description.

Fault Type	Description
Acceptable Value	Parameter is initialized with a correct value.
Missing local variable initialization	Parameter with a random initial value.
Forbidden values	Uses values that are explicitly identified in the DDK documentation as incorrect.
Out of bounds value	Parameters that exceed the expected range of values.
Invalid pointer assignment	Invalid memory locations.
NULL pointer assignment	NULL value passed to a pointer parameter.
Related function not called	This fault is produced by deliberately not calling a setup function, contrarily to what is defined in the DDK documentation.

4.4 Expected Failure Modes

The list displayed in Table 4-5 represents the possible scenarios that are expected to occur after a DD injects a fault into the OS. Initially we started with a much larger list of failure modes, which was derived from various sources, such as the available works in the literature and expert opinion from people that administer Windows systems. However, as the experiences progressed, we decided to reduce substantially this list because several of the original failure modes were not observed in practice.

Generally speaking, there are two major possible outcome scenarios: either the faulty input produces an error (e.g., a crash) or it is handled in some manner. Since the fault handling mechanisms can also have implementation problems, the FM1 failure mode was divided in three subcategories. In order to determine which subcategory applies to a given experiment, the DD verifies the correctness of the return value (if it was different from void) and output parameters of the function.

- **Returns ERROR (RErr):** The return value from the function call indicates that an error was detected possibly due to invalid parameters. This means that the bad input was detected and was handled properly.
- **Returns OK (ROk):** The return value of the call indicates a successful execution. This category includes two cases: even with some erroneous input, the function executed correctly or did not run but returned OK; all input was correct, for instance because only good parameter values were utilized or the random parameters ended up having acceptable values.
- **Invalid return value (RInv):** Sometimes several values are used to indicate a successful execution (a calculation result) or an error (reason of failure). When the return value is outside the range of possible output values (at least from what is said in the DDK documentation), this means that either the documentation or the function implementation has a problem.

Table 4-5: Expected failure modes.

ID	Description
FM1	No problems are detected in the system execution.
FM2	The applications or even the whole system hangs.
FM3	The system crashes and then reboots; the file system is checked and NO corrupted files are found.
FM4	Same as FM3, but there are corrupted files.

The experimental system was configured such that whenever a crash occurs, Windows generated a minidump file to describe the execution context of the system when the failure took place. The analysis of this file is very important because it allows developers to track the origin of crashes. Although several efforts have been made to improve the capabilities of crash origin identification, still some errors remain untraceable or are detected incorrectly.

Whenever an experiment caused a crash, the minidump files were inspected to evaluate their identification capabilities. Four main categories of results were considered:

- **Identification OK (M1):** The minidump file correctly identifies the faulty driver as the source of the crash.
- **Identification ERROR (M2):** The minidump file identifies other module as the cause of failure.
- **Unidentified (M3):** The minidump file could not identify either the driver or other module as the source of the crash.
- **Memory Corruption (M4):** The minidump file detected a memory corruption.

4.5 Experimental Setup

Since the experiments were likely to cause system hangs or crashes, and sometimes these crashes corrupted files, two machines were used to automate most of the tasks (see Figure 4-2). The target machine hosts the OS under test and the DD workload, and the controller machine is in charge of selecting which tests should be carried out, collecting data and rebooting the target whenever needed.

After booting the targeting machine, the DevInject contacts the DevController to find out which driver should be used in the next experiment. Then, DevInject loads the driver, triggers the fault, checks the outcome and, if everything went well, removes the driver.

The DevController is informed of each step of the experiment, so that it can instruct the DevInject what actions should be performed next. This way, the target file system is not used to save any intermediate results or keep track of the experience, since it might end up being corrupted. The target file system is however utilized to store the minidump files and the corrupted files that were found. After a reboot, the DevInject transfers to the DevController all this information using FTP.

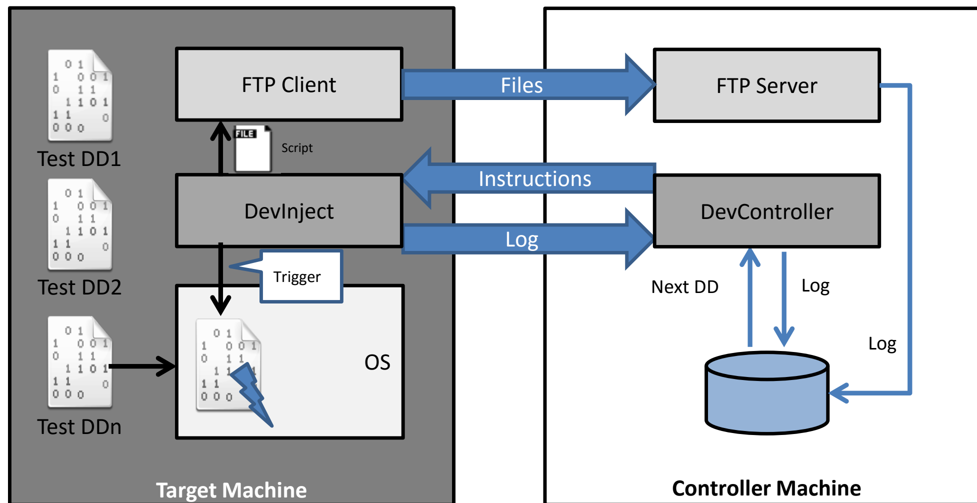


Figure 4-2: Experimental setup.

All measurements were taken on a prototype system composed by two x86 PCs linked by an Ethernet network. The target machine was a DELL Optiplex computer with 512Mb and 2 disks.

Three OS versions and two distinct file systems, FAT32 and NTFS, were evaluated. The outcome was five different configurations (Vista was not tested with FAT32). The exact OS versions were: Windows XP Kernel Version 2600 (SP 2), built: 2600.xpsp_sp2_gdr.050301-1519, Windows Server 2003 Kernel Version 3790 (SP 1), built: 3790.srv03_sp1_rtm.050324-1447 and Windows Vista Kernel Version 5600, built: 5600.16384.x86fre.vista_rc1.060829-2230.

Microsoft provides an equivalent DDK for all OS. This way the same set of drivers that have been synthetically produced could be used to test the various OS. In every target configuration the initial conditions were the same, the OS were configured to produce similar types of dump files, and the DevInject tool was basically the only user application running.

The experiments were performed without load to ensure that results were highly repeatable, and therefore to increase the accuracy to the conclusions.

4.6 Discussion of Results

The observed failure modes are displayed in Table 4-6. The first three columns present the function identifier ID, its alias name and the number of experiments carried out with each function. The failure modes for the various OS configurations are represented in the next four groups of columns, under the headings FM1 to FM4. Each column group presents one value for each OS configuration.

Table 4-6: Observed failure modes.

ID	Alias	#DD	FM1: Execution OK					FM2: Hangs					FM3: Crash					FM4: Crash & FCorrupt				
			XP		2003		V	XP		2003		V	XP		2003		V	XP		2003		V
			FAT	Ntfs	FAT	NtFs	Ntfs	FAT	Ntfs	FAT	NtFs	NtFs	FAT	Ntfs	FAT	NtFs	NtFs	FAT	Ntfs	FAT	NtF	NtFs
1	InitStr	12	9	9	9	9	9	0	0	0	0	0	2	3	2	3	3	1	0	1	0	0
2	AllocPool	440	416	416	416	416	420	0	0	0	0	0	14	24	13	24	20	10	0	11	0	0
3	BugCheck	12	0	0	0	0	0	0	0	0	0	0	6	12	3	12	12	6	0	9	0	0
4	CompReq	51	0	0	0	0	0	0	0	0	0	0	0	51	26	51	51	51	0	25	0	0
5	CreateDev	96	48	48	48	48	48	0	0	0	0	0	29	48	8	48	48	19	0	40	0	0
6	DeleteDev	4	0	0	0	0	0	0	0	0	0	0	0	4	2	4	4	4	0	2	0	0
7	InitEvt	18	6	6	6	6	6	0	0	0	0	0	8	12	7	12	12	4	0	5	0	0
8	WaitObj	36	18	18	18	18	18	0	0	0	0	0	15	18	2	18	18	3	0	16	0	0
9	ZwClose	3	3	3	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	CallDrv	9	0	0	0	0	0	0	0	0	0	0	0	9	4	9	9	9	0	5	0	0
11	FreePool	16	1	1	1	1	1	0	0	0	0	0	1	15	8	15	15	14	0	7	0	0
12	SetEvt	24	6	6	18	18	9	0	0	0	0	0	15	18	5	6	15	3	0	1	0	0
13	InitLock	3	2	2	2	2	2	0	0	0	0	0	0	1	0	1	1	1	0	1	0	0
14	AcqLock	8	0	0	0	0	0	2	2	2	2	2	2	6	2	6	6	4	0	4	0	0
15	RelLock	48	3	3	1	1	1	0	0	0	0	0	18	45	12	47	47	27	0	35	0	0
16	DerefObj	3	2	2	2	2	0	0	0	0	0	0	1	1	0	1	3	0	0	1	0	0
17	OpenKey	155	104	104	104	104	104	0	0	0	0	0	25	51	47	51	51	26	0	4	0	0
18	QryKey	315	315	315	315	315	315	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	AttachDev	9	0	0	0	0	0	1	1	1	1	1	6	8	2	8	8	2	0	6	0	0
20	memset	48	18	18	27	27	24	0	0	0	0	0	22	30	12	21	24	8	0	9	0	0
Total		1310	951	951	970	970	960	3	3	3	3	3	164	356	155	337	347	192	0	182	0	0
Total / # DD (%)			72,6	72,6	74,0	74,0	73,3	0,2	0,2	0,2	0,2	0,2	12,5	27,2	11,8	25,7	26,5	14,7	0	13,9	0	0

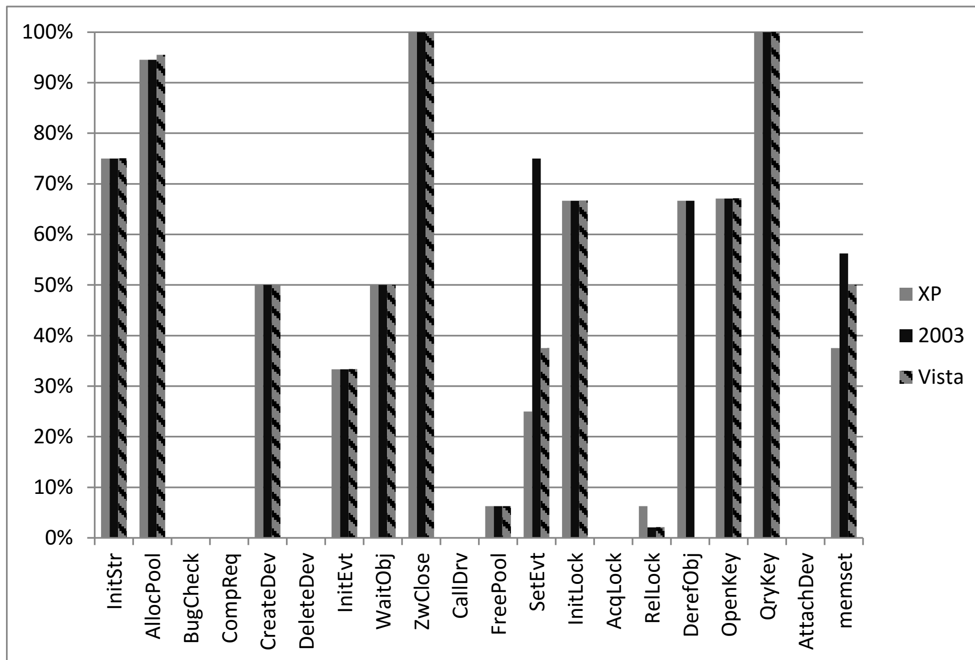


Figure 4-3: Relative robustness (FM1/#DD).

In the 20 functions that were tested, several of them were able to deal at least with a subset of the erroneous input. There were however a few cases where results were extremely bad, indicating a high level of vulnerability.

By computing the formula FM1/#DD for each FM1 entry, one can have an idea about the relative robustness of the functions (see Figure 4-3). The results obtained using Windows XP with FAT and NTFS files systems were the same. This also happened in the case of Windows 2003. For these reason, we are showing a more simplified view of the results. As displayed in the graph, only two functions were 100% immune to the injected faults, ZwClose and QryKey. On the other hand, eight functions had zero or near zero capabilities to deal with the faults.

One reason for this behaviour is that some of these functions are so efficiency dependent (e.g., CompReq and AcqLock) that developers probably have avoided the implementation of built in checks. Another reason is related to the nature of the function, which in the case of BugCheck is to bring down the system in a controlled manner, when the caller discovers an unrecoverable inconsistency. In this case, the developers probably preferred to reboot the system even if some parameters were incorrect (but notice that this reboot sometimes was not done in a completely satisfactory way since files ended up being corrupted).

Observing again Table 4-6 from the various functions it is possible to conclude that only two caused the system to hang (vertical section FM2: Hangs). Functions AcqLock and AttachDev caused hangs in all OS configurations, when an invalid pointer was passed as argument. Most of the erroneous inputs that caused failures end up crashing the system (vertical section FM3 and vertical section FM4). From the various classes of faults that were injected, the most malicious were invalid pointer assignments and NULL values passed in pointer parameters. The first class, invalid pointers, is sometimes difficult to validate, depending on the context (e.g., a buffer pointer that was not properly allocated but has a different value than NULL). On the other hand, NULL pointers can be easily determined and for this reason it is difficult to justify why they are left un-checked, allowing them to cause so many reliability problems.

In all experiments, it was never observed any file corruption with the NTFS file system after a reboot. However, the FAT32 file system displayed in many instances cases of corruption. Traditionally, NTFS has been considered much more reliable than FAT32, and our results contribute to confirm this. The reliability capabilities integrated in NTFS, like transactional operations and logging, have proven to be quite effective at protecting the system during abnormal execution. The overall comparison of the 3 operating systems, if we restrict ourselves to NTFS or FAT32, shows a remarkable resemblance among them.

The last two rows of Table 4-6 present an average value for the failure modes and OS configurations. On average, OSs had an approximately equivalent number of failures in each mode, with around 73% testes with no problems detected during the system execution. Hangs were a rare event in all OSs. If a finer analysis is made on a function basis (see Figure 4-3), we observe a similar behaviour for most functions. There were only two functions where results reasonably differ, SetEvt and memset. From these results, there is reasonable indication that the 3 operating systems use comparable levels of protection from faulty inputs coming from drivers.

These results reinforce the idea that although the Windows NT system has undergone several name changes over the past several years, it remains entirely based on the original Windows NT code base. However, as time went by, the implementation of many internal features has changed. We expected that newer versions of the Windows OS family would become more robust; in practice, we did not see this improvement at the driver's interface. Of course, this conclusion needs to be better verified with further experiments.

Return Values from Functions

As explained previously, even when the system executes without apparent problems, the checking mechanisms might not validate the faulty arguments in the most correct manner and produce fail-silent violations. Therefore, FM1 can be further divided in three sub-categories to determine how well the OS handled the inputs.

Table 4-7 shows the results of the experiments obtained when the function execution returned a value in the RErr category, i.e., an error was detected by the function. Since some functions do not return any values, their corresponding table entries were filled with "-". The "# Faulty Drivers" column refers to the number of drivers produced by DevBuilder that contained at least one bad parameter. Comparing this column with the following five columns, one can realize that only two functions have a match between the number of faulty drivers and the number of RErr values. The other functions revealed a limited parameter checking capability.

Table 4-7 Return error (RErr) values.

ID	Alias	#Faulty Drivers	RErr				
			XP		2003		Vista
			Fat	Ntfs	Fat	Ntfs	Ntfs
1	InitStr	9	0	0	0	0	0
2	AllocPool	200	20	20	20	20	12
3	BugCheck	12	-	-	-	-	-
4	CompReq	51	-	-	-	-	-
5	CreateDev	76	0	0	0	0	0
6	DeleteDev	4	-	-	-	-	-
7	InitEvt	14	-	-	-	-	-
8	WaitObj	36	0	0	0	0	0
9	ZwClose	3	3	3	3	3	3
10	CallDrv	9	0	0	0	0	0
11	FreePool	15	-	-	-	-	-
12	SetEvt	20	0	0	0	0	0
13	InitLock	2	-	-	-	-	-
14	AcqLock	8	0	0	0	0	0
15	RelLock	48	-	-	-	-	-
16	DerefObj	3	-	-	-	-	-
17	OpenKey	155	104	104	104	104	104
18	QryKey	315	315	315	315	315	315
19	AttachDev	9	0	0	0	0	0
20	memset	39	0	0	0	0	0

To complement this analysis, Table 4-8 presents the results for the ROk category (i.e., the return value of the call is a successful execution). Column "Non Faulty Drivers" shows the number of drivers with only correct arguments. Comparing this column with the remaining ones, it is possible to conclude that functions return a successful execution more often than the number of non-faulty drivers. However, in some cases this might not mean that there is a major problem. For instance, consider function 2-AllocPool that receives three parameters: the type of pool (P0); the pool size (P1); and a tag value (P2). Depending on the order of parameter checking, one can have the following acceptable outcome: P1 is zero, and 2-AllocPool returns a pointer to an empty buffer independently of the other parameters values.

On the other hand, by analysing the execution log, we found out that when P1 was less than 100.000*PAGE_SIZE, Windows returned ROk even when a forbidden value was given in P0 (at least, as stated in the DDK documentation). This kind of behaviour means that an error was (potentially) propagated back to the driver, since it will be using a type of memory pool different from the expected thus causing a fail silent violation. The table also reveals another phenomenon -- the three versions of Windows handle the faulty parameters differently.

Table 4-8 Return OK (ROk) values.

ID	Alias	Non Faulty Drivers	ROk				
			XP		2003		Vista
			Fat	Ntfs	Fat	Ntfs	Ntfs
1	InitStr	3	9	9	9	9	9
2	AllocPool	240	396	396	396	396	408
3	BugCheck	0	-	-	-	-	-
4	CompReq	0	-	-	-	-	-
5	CreateDev	20	48	48	48	48	48
6	DeleteDev	0	-	-	-	-	-
7	InitEvt	4	-	-	-	-	-
8	WaitObj	0	18	18	18	18	18
9	ZwClose	0	0	0	0	0	0
10	CallDrv	0	0	0	0	0	0
11	FreePool	1	-	-	-	-	-
12	SetEvt	4	6	6	18	18	9
13	InitLock	1					
14	AcqLock	0	0	0	0	0	0
15	RelLock	0	-	-	-	-	-
16	DerefObj	0	-	-	-	-	-
17	OpenKey	0	0	0	0	0	0
18	QryKey	0	0	0	0	0	0
19	AttachDev	0	1	1	1	1	1
20	memset	9	18	18	27	27	22

For example, there were several cases in Vista where function 2-AllocPool succeeded while in XP and Server 2003 it caused a crash. In function 12-SetEvt, Server 2003 does not crash when TRUE was passed in one of the parameters, while the other did so (the documentation says that when this value is used, the function execution is to be followed immediately by a call to one of the KeWaitXxx routines, which was not done in either OSs).
In all experiments, we did not observe any return values belonging to the RInv category (i.e., values outside the expected return range).

Corrupted Files

The last group of results in Table 4-6 corresponding to FM4, displays the number of times Windows found corrupted files while booting. The Chkdsk utility is called during the booting process to detect these files.

Corrupted files were found only in the configurations that used the FAT32 file system. Using the formula FM4/(FM3+FM4) one can have a relative measure of how sensitive is the file system when a crash occurs, i.e., Σ crashes resulting in corrupt files / Σ crashes. The results presented in Figure 4-4 shows that when using FAT32 in general, Windows Server 2003 is more sensitive than Windows XP in a majority of the cases (since there were no observed crashes for Windows XP using NTFS, Windows Server 2003 NTFS and Windows Vista these results were omitted from the graph for simplicity).

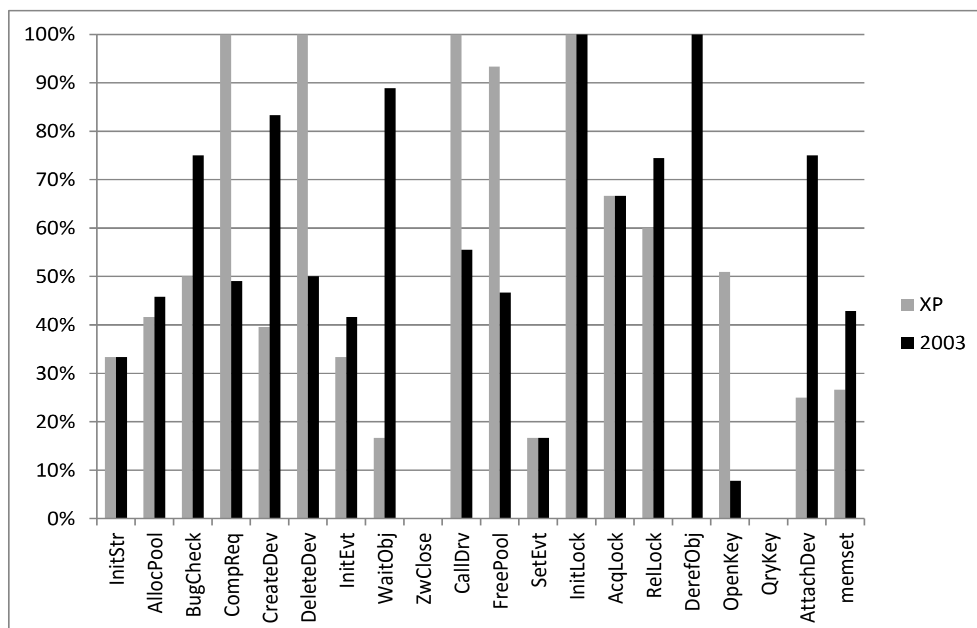


Figure 4-4: File System sensitiveness (FM4/(FM3+FM4)).

Minidump Diagnosis Capabilities

The analysis of the minidump files produced during a system crash allows us to determine how well they identify a driver as the culprit of the failure. These files are fundamental tools for the Windows development teams because they help to diagnose system problems, and eventually to correct them. We have used the Microsoft's Kernel Debugger [101] to perform the analysis of these files, together with a tool, DevDump, that automates most of this task. DevDump controls the debugger, passes the minidumps under investigation, and selects a log where results should be stored. After processing all files, DevDump generates various statistics about the detection capabilities of minidumps.

In the experiments, all Windows versions correctly spotted the faulty DD in the majority of times. Figure 4-5 show the relationship between the number of crashes and the correct identification of the source of the crash (M1). The accuracy of the error source determination seems to be independent of the file system used. Only in very few cases there was a difference between the two file systems, such as for the 7-InitEvt function where Server 2003 FAT32 identified a different source of crash from Server 2003 NTFS. In general, the results show that Windows XP is more accurate than the others OS (see 7-InitEvt, 14-AcqLock and 15-RelLock). However, there were cases where other kernel modules were incorrectly identified (functions 1-InitStr, 14-AcqLock and 15-RelLock).

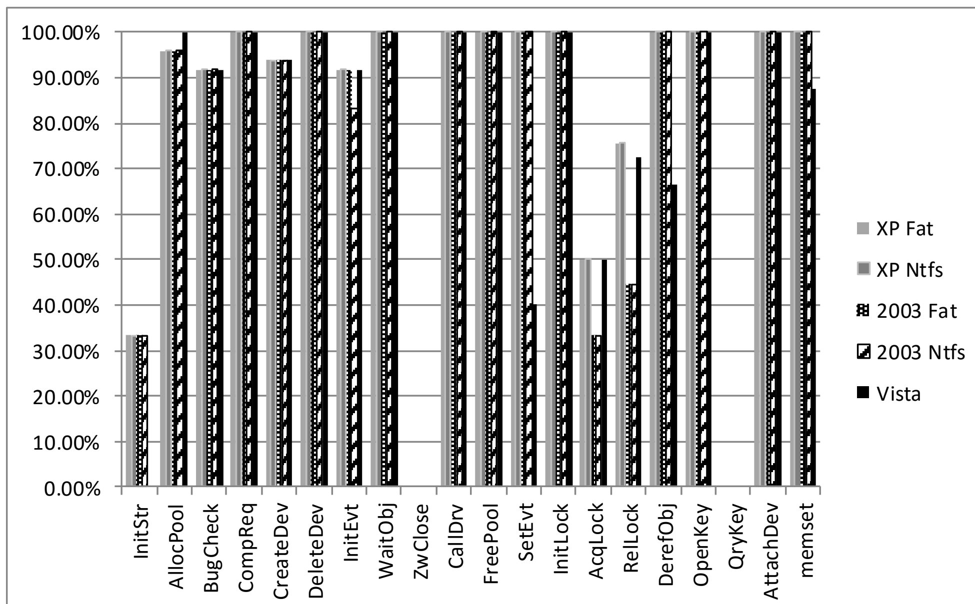


Figure 4-5: Source identification OK (M1).

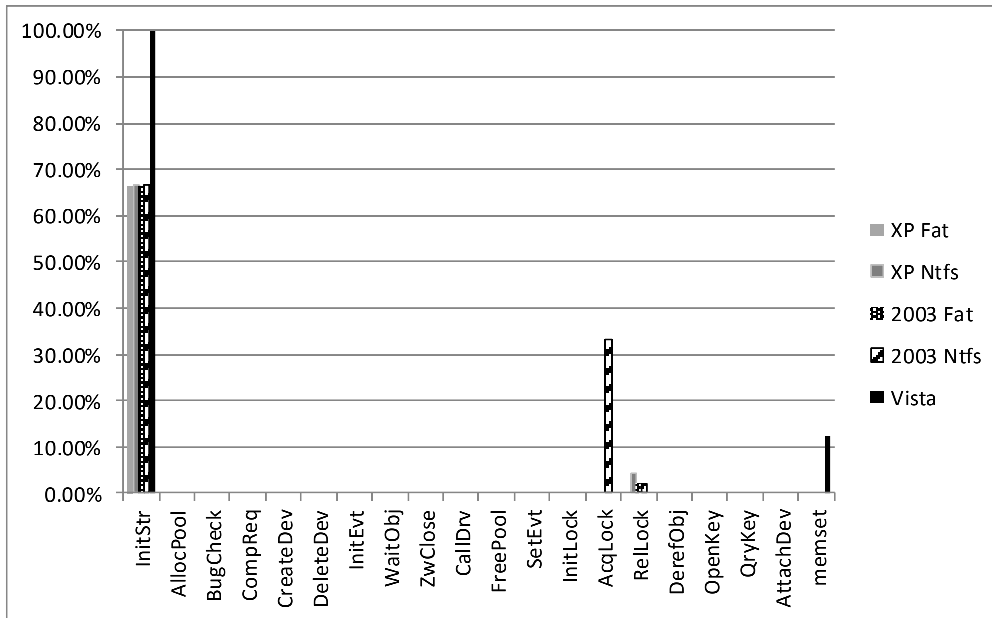


Figure 4-6: Source identification error (M2).

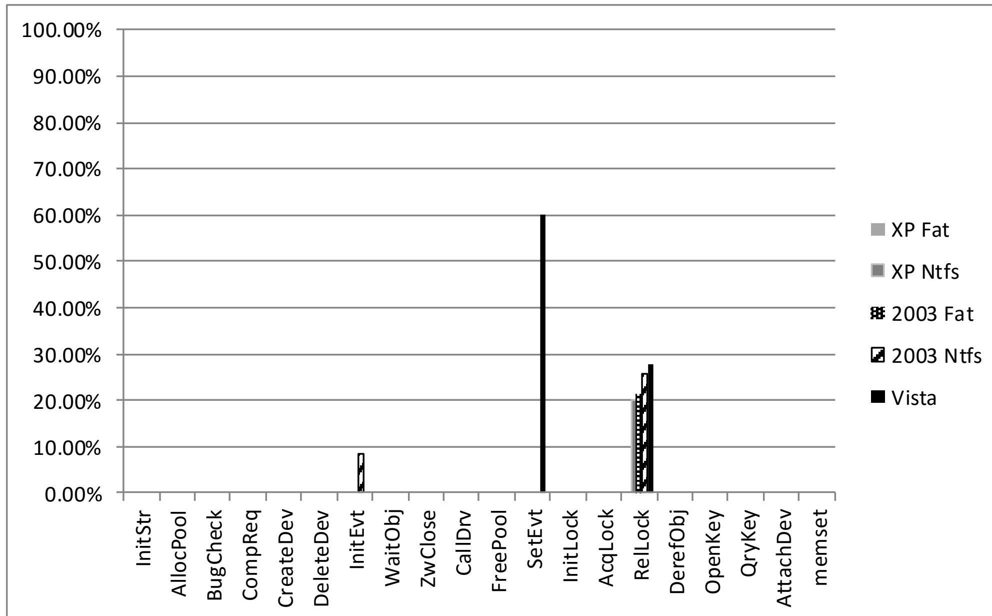


Figure 4-7: Source of crash unidentified (M3).

These errors are particularly unpleasant because they can lead to waste of time while looking for bugs in the wrong place, and they can reduce the confidence on the information provided by minidumps. In some other cases, Windows was unable to discover the cause of failure. This happened in Vista more frequently than the other OS configurations, for instance in functions 12-SetEvt and 15-RelLock (see

Figure 4-7). In function 12-SetEvt, Vista was the only system that could not diagnose the cause of failure. Only Windows Server 2003 detected memory corruption situations (in functions 14-AcqLock and 15-RelLock). Windows Server 2003 (FAT32 and NTFS) located memory corruptions when faults were injected in functions 14-AcqLock and 15-RelLock.

4.7 Summary

This investigation focused on a robustness testing experiment that evaluates Windows XP, Windows Server 2003 and Windows Vista. The main objective of this study was to determine how well Windows protects itself from faulty drivers that provide erroneous input to the DDK routines. Seven classes of typical programming bugs were simulated.

The analysis of the results shows that most interface functions are unable to completely check their inputs - from the 20 selected functions, only 2 were 100% effective in their defence. We observed a small number of hangs and a reasonable number of crashes. The main reason for the crashes was invalid or NULL pointer values. Corruption of files was only observed with the FAT32 file system. The analysis of the return values demonstrates that in some cases Windows completes without generating an error for function calls with incorrect parameters, in particular, Windows Server 2003 seems to be the most permissible one. This behaviour suggests a deficient error containment capability of the OS. In most cases, the examined minidump files provided valuable information about the sources of the crashes, something extremely useful for the development teams. However, Windows Vista seems to have more troubles in this identification than the other OS. The experiments made with Windows Vista revealed that it behaves in a similar way to Windows XP and Server 2003.

CHAPTER 5 ATTACKING WI-FI DRIVERS

WLAN were originally employed to provide networks elements with the ability to roam across facilities. They give individuals the freedom to stay connected to the network while moving from one coverage area to another. They can be used to extend a wired infrastructure or to replace the existing ones, and save costs not only due to the falling price of the wireless components but primarily with savings with power and data cables installation.

WLAN offer many advantages but also weaken the security perimeter. In many places, like airports and shopping malls, there are dozens of rogue networks just waiting to entrap unsuspecting travellers. Every time someone logs on to a public WLAN, it is transmitting its login name and password over open airwaves, and when accessing the Internet possibly its credit card number.

Individual home networks may be attractive to malicious neighbours wanting to steal the bandwidth or passers-by snooping around one's hard disk. Corporate networks may be of increased interest to hackers willing to steal business secrets, credit card transactions, personal data or health care records. This happens because many public and private WLAN use poor or no encryption at all, meaning

that anyone with a laptop and a WLAN card could intercept and read data packets being sent or received by legitimate users.

Although many security failures are due to incorrect configuration, some are caused by implementation errors. In this chapter, we are particularly interested in locating this sort of bugs (or vulnerabilities) in DD of WLAN, to allow their removal. In the majority of situations, the code of the DD is closed. Therefore, the most common way for vulnerabilities to be discovered by hackers is to use a black box testing methodology using random inputs, sometimes called fuzzers [63]. It consists on presenting malformed data injection to the interface and observe the outcomes. This technique may require further refinements to catch more complex bugs, due to protocol specificities, but it can be very effective discovering most obvious ones, like TCP-IP stack problems and OS hangs.

This chapter presents the design of a new fuzzer architecture that is able to build malformed packets and perform attacks against target systems, independently of the communication media. The current implementation of the architecture, called Wdev-Fuzzer, supports the Wi-Fi protocol but it can be extended to other communication protocols, such as IrDA and Bluetooth. The tool was utilized to study the behaviour of a Wi-Fi DD of a smart phone running Windows Mobile 5. The tested scenarios simulate an attack against the Wi-Fi device, either when it is just looking for an Access Point (AP) to connect or when it is already connected.

Experimental results demonstrated that in most cases Windows is capable of handling correctly the malicious packets. However, in one situation, a specific Beacon packet always caused the system to hang. This implies that the DD has a critical vulnerability which was previously unknown. Wdev-Fuzzer was also successfully applied to uncover other potential problems. For example, it was used to reproduce denial of service attacks with Disassociation and Deauthentication frames.

5.1 Wdev-Fuzzer Architecture

The Wdev-Fuzzer is divided in 8 modules (see Figure 5-1). The Message Specification is a text file that defines packets as a group of fields. Each packet field is also specified in the same file using basic data types that are intrinsic to the Wdev-Fuzzer. For each basic type there is a fuzz operator that assigns specific values according to some given rules. During the construction of the packets, the Packet Generator takes the packet description as input, and uses these operators to fill in the values of the fields.

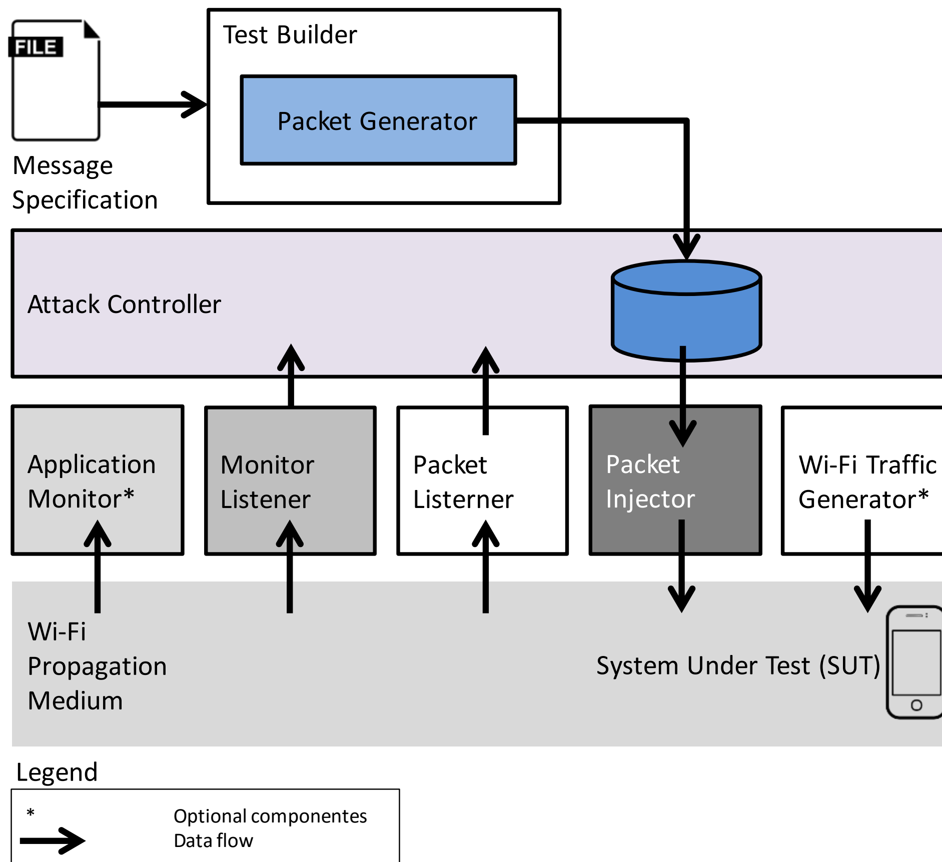


Figure 5-1: Wdev-Fuzzer block diagram.

The result is a ready-to-be-send potentially bogus packet. By extending the basic types and the fuzz operators, it is possible to build newer types and values, in order to meet specific protocol requirements.

The Packet Injector sends the packets to the SUT. And the Packet Listener receives and analyses all responses that arrive from the SUT. The Monitor Application and corresponding Monitor Listener are optional components that exchange information about the state of the SUT. They are used to help to find out if an attack was successful and contribute to the decision of which attack should be performed next. The Attack Controller controls the activity of the Packet Injector. It decides which next packet (attack) should be transmitted, based on the feedback given by the Monitor Listener and Packet Listener, using predetermined criteria.

The Traffic Generator is used to create and exchange good packets between the Access Point (AP) and the SUT. This way we can observe the system behaviour when subject to an attack while correct data is being transmitted by a non-malicious AP.

The basic architecture of Wdev-Fuzzer can be tailored to several communication protocols, still some changes will have to be performed. For example, a new Message Specification has to be carried out and the Packet Injector and Packet Listener implementations have to be updated to use the specific functions for sending and receiving raw packets from the media.

5.2 Using Wdev-Fuzzer in 802.11

The IEEE 802.11 architecture consists of several interacting components to provide a WLAN that supports station mobility transparently to upper layers. The basic service set (BSS) is the fundamental building block of an IEEE 802.11 LAN. The BSS coverage area is where the member stations (STA) of the BSS may remain in communication. If a STA moves out of its BSS, it can no longer directly communicate with the other members.

The independent BSS (IBSS) is the most basic type of a Wi-Fi LAN, and consists of only two STA that are able to exchange data directly with each other. Since this type of network is often formed without pre-planning it is usually referred to as an ad-hoc network.

A BSS, instead of operating independently, may also be part of an extended form of network that is built with multiple BSSs and is interconnected by a distribution system (DS). In this setting, an AP gives access to the DS by providing DS services in addition to act as a STA.

Figure 5-2 shows the Medium Access Control (MAC) message frame format for the 802.11 protocol. These frames may be composed by Fixed Length (FL) and Tag Length Value (TLV) field types.

To facilitate message parsing, when FL and TLV fields appear in the same message, FL fields always come first. A FL field appears at a fixed location relative to the beginning of the frame and it always has the same length. A TLV field has three elements, a Tag which uniquely identifies the field, a size element which determines the length of the data and the data itself.

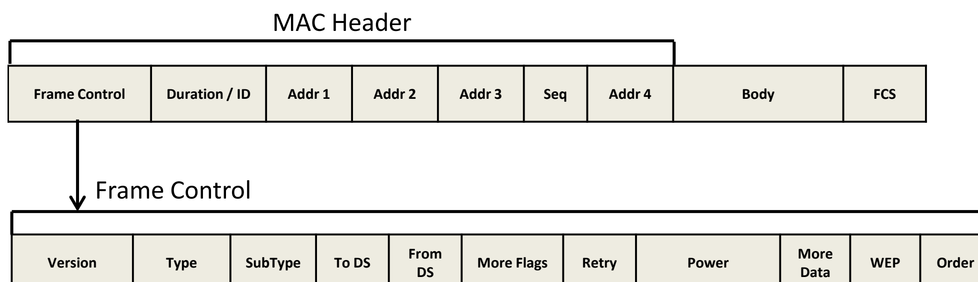


Figure 5-2: Generic Wi-Fi MAC frame format.

Examples of FL fields are all the contents of the Frame Control. Examples of TLV fields are for instance the Traffic Information Map (TIM) field in a Beacon frame.

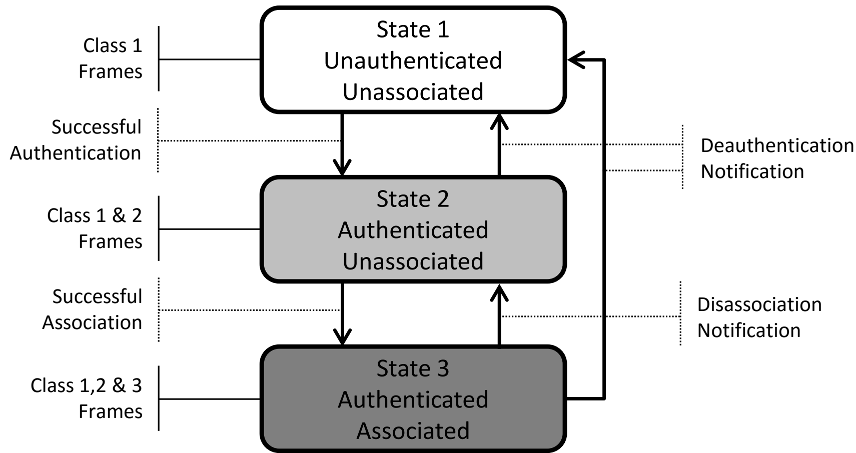


Figure 5-3: Relationship between messages and services in Wi-Fi.

Table 5-1: Tested Wi-Fi frames.

Frame	Type	SubType	To AP	From AP	Class
Association Request	Mgt	0	✓	-	2
Association Response	Mgt	1	-	✓	2
Reassociation Request	Mgt	2	✓	-	2
Reassociation Response	Mgt	3	-	✓	2
Probe Request	Mgt	4	✓	-	1
Probe Response	Mgt	5	-	✓	1
Beacon	Mgt	8	-	✓	1
Disassociation	Mgt	10	✓	✓	2
Authentication	Mgt	11	✓	✓	1
Deauthentication	Mgt	12	✓	✓	1,3
Power Save	Ctrl	10	✓	-	3
Request to Send	Ctrl	11	✓	-	1
Clear to Send	Ctrl	12	-	✓	1
Acknowledgment (Ack)	Ctrl	13	✓	✓	1
Contention Free (CF) End	Ctrl	14	-	✓	1
CF-End+CF-Ack	Ctrl	15	-	✓	1
Data	Data	0	✓	✓	1,3

✓ Field included in the message

The MAC frame types that may be exchanged between a pair of STAs depend on their state. The state of the sending STA, given by Figure 5-3, is defined with respect to the intended receiving STA. The allowed frame types that can be transmitted in a given state are grouped into classes. In State 1, only Class 1 frames are allowed. In State 2, either Class 1 or Class 2 frames are acceptable. In State 3, all frames are permitted (Classes 1, 2, and 3). The frame classes are shown in Table 5-1.

In this work, we utilize the Wdev-Fuzzer to evaluate the Wi-Fi implementation of a Windows Mobile 5 smart phone. Since these type of equipment are mostly used as a STA rather than as an AP, the device will be configured as an STA. The evaluation of an AP is left out for future work. Additionally, we will not use the IBSS configuration because handheld devices are many times operated in a connected BSS. In the tested scenarios, the Wdev-Fuzzer is going to simulate a malicious AP that sends potentially erroneous frames to a SUT.

Table 5-2: Tested Faulty Values.

Fuzz Operator	Fixed Length Field	Tag Length Value Field
Not Present	-	✓
Repeated	-	✓
All bits Zero	✓	✓
MIN-1	✓	✓
MIN	✓	✓
MIN+1	✓	✓
Random	✓	✓
Specific Value	✓	✓
MAX-1	✓	✓
MAX	✓	✓
MAX+1	✓	✓
All bits One	✓	✓

✓ Tested condition

5.3 Tested Faulty Values

Table 5-2 displays the fuzz operators that are applied to each field type, to build Wi-Fi frames in the experiments. The '✓' character indicates that the operator was applied to the field and the '-' the opposite. The operator "Not present" omits an element from the frame. The "Repeated" operator produces multiple occurrences of the same field in the frame. The operators "All bits Zero" and "All bits One" are self-

explanatory. The “MIN” and “MAX” operators produce the minimum and maximum values that a field might contain, as stated in the 802.11 specification.

Often, the “All bits Zero” and “MIN” operators produce equal values, whenever the minimum value is zero. The same applies for operators “MAX” and “All bits One”. In these cases, the “MIN” or “MAX” operators are not utilized, since they create test results equivalent to the “All bits Zero” and “All bits One” (respectively).

The “Random” operator generates random values that are between the values produced by the “MIN” and “MAX” operators. At last, the “Specific Value” operator places a pre-defined value in a field. This operator is used for example to force certain frames to have SUT’s MAC address.

5.4 Tested Scenarios

At first, we considered testing the SUT in all 3 states represented in Figure 5-3. However, since in real situations State 2 is only available for shorts periods of time, only States 1 and 3 were considered.

Tests were carried out in 3 different scenarios (A, B and C). In scenario A, the SUT was in State 1, meaning that it was not associated or authenticated with any AP. In scenario B, the SUT was in State 3, linked to a Real AP using no authentication. At last, in scenario C, the SUT was also at State 3 but using authentication. In scenarios B and C, the Traffic Generator forced the exchange of data packets between the SUT and the Real AP to stress the communication stack by opening a TCP-IP socket and transmitting packets between the SUT and the Real AP.

5.5 Expected Failure Modes

The Packet Generator uses the Message Specification and the fuzz operators to build Wi-Fi frames. Depending on the values produced, the SUT is going to receive good and bad Wi-Fi frames, which may be handled correctly or may lead to some failure. Table 5-3 summarizes the expected failure modes of the SUT when it receives Wi-Fi frames. It was elaborated after some preliminary experiments and also based on information provided in the literature [53][54].

F1 represents the case where the system appears to continue to work without any problems. However, in general, it does not mean that the injected fault was handled correctly. Whenever a test uses Beacon or Probe frames, the SUT Monitor returns some feedback to the Controller, saying which APs have been detected.

Table 5-3: Expected failure modes.

ID	Description
F1	No problems were detected in the system execution.
F2	Packet Listener detects invalid frame.
F3	SUT was disassociated.
F4	SUT was de-authenticated.
F5	Monitor hangs.
F6	OS hangs.
F7	The system crashes and then reboots.

Table 5-4: Detailed F1 failure mode.

ID	Description
F1A	Device provides correct information about AP (either detecting it or not).
F1B	Device does not detect the AP but it should.
F1C	Device detects the AP but it should not.

In these cases, we are able to further extend F1 in three other categories, as represented in Table 5-4. For instance, the F1A value represents the scenario when the Monitor correctly reports the information about the AP, either because it was detected (the packet was well-formed) or because it was not detected (the packet was incorrectly formed, and therefore, the SUT discarded it and the report indicates no AP). The F1B value applies to the cases where the Monitor does not detect the AP but it should, and F1C corresponds to the cases where the AP is detected but it should not.

The F2 failure mode represents the situations where the SUT detected an invalid frame.

When the SUT is at State 3, the F3 failure mode means that the device became disassociated from the AP, as a result of some attack. Likewise, the F4 mode indicates that the attack successfully deauthenticated the SUT from the AP.

The F5 failure mode signals that the Monitor Application hangs as a consequence of an attack, denoting that some problem with the DD has propagated to the application. Whenever the OS hangs, the F6 mode is used. The F7 failure mode corresponds to the situation when the system crashes and then reboots.

5.6 The Testing Infra-structure

In the Windows OS family, NDIS is an API for Network Interface Cards (NIC's). The details of a NIC hardware implementation can be wrapped by a Media Access

Controller (MAC) DD, in such a way that all NIC's for the same media (e.g., Ethernet) are accessed using a common API. Applications interact with NIC's through a stack of DDs, where each driver adds functionality to the entire communication infrastructure.

Probably, the main difficulty in building a Wi-Fi test infrastructure is the implementation of the operations for injecting and capturing the Wi-Fi raw frames. Our first attempt to address the problem utilized a filter DD that was placed in the lower parts of the driver stack, hoping to intercept packets sent and received by each NIC (as well as control instructions given by the OS to the DD). Windows, however, implements the Wi-Fi protocol in the MAC DD, which emulates the Ethernet protocol to the drivers above it. Therefore, the DD was only able to capture Ethernet frames and not Wi-Fi raw frames.

Still there are other possible ways for capturing Wi-Fi frames in Windows, neither of them very easy to achieve. One approach is using an internal interface to the MAC DD. Another consists in developing our own MAC DD, but this would require a direct interaction with the NIC and complete knowledge of its specification (something that usually is not available). A commercial solution based on this idea is Airpcap [64], which uses a proprietary MAC DD and their own capture hardware.

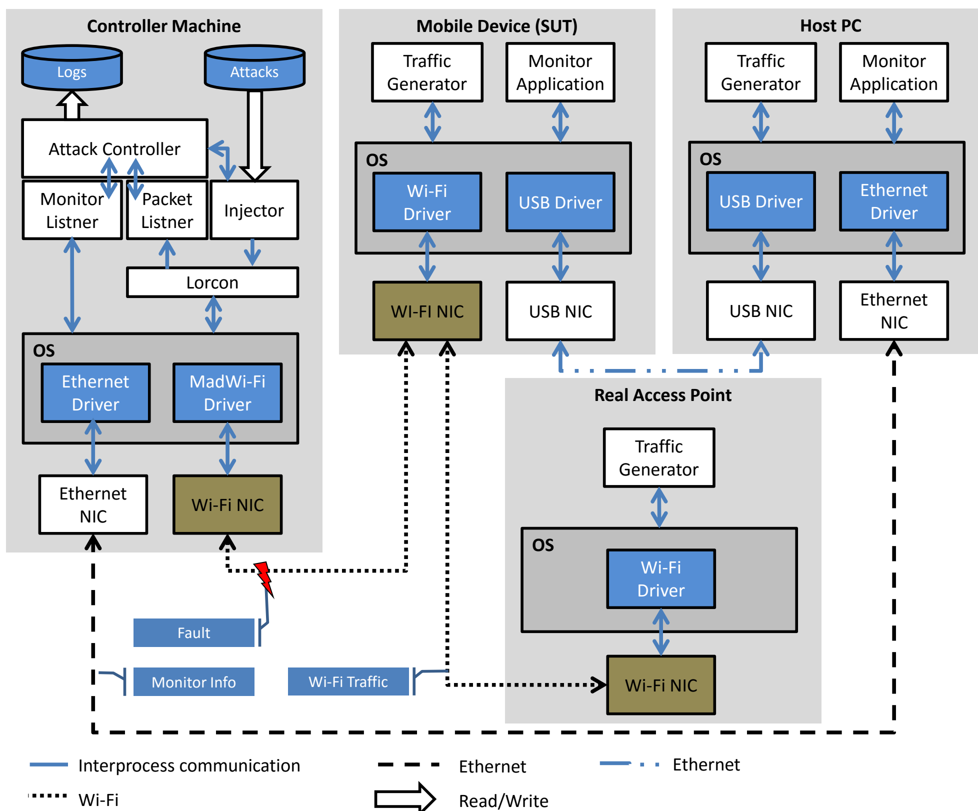


Figure 5-4: Fuzzer Wi-Fi test infrastructure.

In the end, it was decided to build a heterogeneous testing infrastructure, since in Linux there are several cards and open drivers that support Wi-Fi frame injection and capture (although not every NIC can be used due to hardware limitations). One simple way to find them is to search in the Internet for Wi-Fi sniffers and look for compatible NICs.

Figure 5-4 displays the current testing infrastructure that is composed by 4 components: the Controller Machine, the Mobile Device (SUT), the Host PC and the Real Access Point. We will detail these components in the next sections.

Controller Machine and SUT

The Controller Machine generates the Wi-Fi packets containing malicious data (e.g., out-of-bound values, repeated tags) and sends them through the Wi-Fi interface to the SUT. Each packet is sent several times to assure that the SUT is able to receive it.

This element also monitors the outcomes of the tests, and saves the collected data in the disk for future analysis. Currently, the Controller is installed in a Linux OS machine, with the MadWi-Fi driver [65] for wireless LAN chipsets from Atheros. The Packet Injector uses a modified version of Lorcon [66] as a generic library for injecting Wi-Fi frames. The Monitor Listener receives any incoming frames from the Monitor installed in the SUT and forwards this information to the Attack Controller to synchronize the next attack. The Packet Listener informs the Attack Controller of each incoming packet sent by the SUT. These packets have to be carefully examined to detect any unexpected behaviour.

The SUT is the target Wi-Fi device of the experiments. It runs a Monitor Application that regularly connects to the Monitor Listener of the Controller, informing the current list of detected AP and the status of any existing connection. This data is especially useful when testing Beacon and Probe frames, as the detection of the AP is crucial to determine the correction of the error handling mechanisms.

Host PC and Real AP

The SUT is physically attached to the Host PC through an USB port. This way, the Monitor Application can reach the Attack Controller through an out of band link, leaving the Wi-Fi medium free for the experiments. The Host PC runs Windows XP and Microsoft's ActiveSync, allowing the communication between the SUT and the Host PC with TCP over USB, which is then followed by TCP over Ethernet in the connection between the Host PC and the Controller Machine.

To keep the complexity of the code of the Controller manageable, a Real AP is utilized to take the SUT through the various states of the Wi-Fi protocol. This way, specific frames can be injected in every state. The Real AP was implemented in Windows XP using an off-the-shelf AP application.

5.7 Experimental Results

This section presents the results of the various experiments carried out with the Wdev-Fuzzer in an 802.11b network. The test bed was composed by a Controller Machine implemented in a Dell Optiplex 170L Pentium IV computer, installed with Fedora Core 6. It used a NetGear WPN311 wireless PCI card and the built-in Ethernet card as communication means.

The SUT was an HP iPAQ hw6915 PDA running Windows Mobile 5 and equipped with a built-in Texas Instruments Wi-Fi chip. The Host PC machine was a HighScreen Pentium IV computer with Windows XP Professional Edition. The SUT was attached to an USB port on the Host and uses ActiveSync 4.1 build 4841 to establish the connection. This machine was also equipped with an Ethernet card, which was connected to the Controller Machine with a 100Mbits link. It also hosts the Real AP using a GigaByte AirCruiser GN-WP01GS wireless PCI card and the companion AP application. The SUT was attached to an USB port on the Host PC and placed at about 2m distance from the Controller Machine and the Real AP.

The results of the test campaigns are displayed in Table 5-5 and Table 5-6. A total of 89489 attacks were carried out for each of the three scenarios. The tables only show the outcomes for frames that flow from the AP to the SUT (see Table 5-1), since frames on the other direction never caused any problems (i.e., the failure mode was always of type F1). The first column of the tables shows the field type being tested, and the second column displays how many different values were tried. The following columns display the results obtained for the various different frames. An empty cell is used to indicate that the corresponding field does not belong to the frame being tested otherwise it is filled with the code of the observed failure mode (see Table 5-3).

Since in most cases the result was F1, to make the table reading simpler, the number of times that it occurs is omitted (it is equal to number of tried values displayed in the second column). For failure modes different than F1, the table presents in the cell the number of tests that caused a problem.

Failure Modes in Scenario A

The SUT is in State 1 in the test campaign of scenario A. The SUT is placed in this state by powering on the Wi-Fi component of the device and by making sure that no association exists with any STA or AP. The test results for this scenario are displayed in Table 5-5. It shows that in general the SUT was able to handle correctly the malicious frames. Nevertheless, some interesting outcomes were observed for certain specific scenarios, which are summarized in the following points.

Since Beacon frames are directed to everybody in the coverage area, APs should announce themselves using the broadcast MAC address (`FF:FF:FF:FF:FF:FF`) as the Destination Address. Windows Mobile, however, reports a new AP when the Destination Address uses a distinct MAC address (see row DA). This occurs even when the Destination Address is different from the MAC address of the SUT. This behaviour is an implementation issue and does not seem to be a problem.

SSID is the identifier of the AP, and it has a maximum size of 32 characters. The experiments show that the SUT does not report an existing AP if the SSID field has '0x00' as one of the ASCII characters of the identifier (see row SSID). The same behaviour was also seen when we run an equivalent test with another Windows Mobile equipment, which gives evidence that this problem may extend to several other implementations. From a security perspective, this behaviour is undesirable since it allows the creation of networks which are hidden from certain devices (e.g., a group of hackers could keep a network secret if they found out that the security officers use a Windows Mobile-based solution for diagnosing Wi-Fi networks).

When multiple SSID fields are sent in a given frame, the SUT assumes the last value as the correct one. If other vendors take a different view, and choose for instance the first SSID, then this could lead to incompatibility problems. The 802.11 specification does not address this particular issue.

Whenever the SUT receives a Beacon frame with a TLV field with TAG = 5 (TIM), Length = 255 and Value = 0xFF, the OS hangs at the first user interaction with the device (see F6 value in row TIM). The same kind of failure also occurred when the SUT was in States 2 and 3, as shown in Table 5-6. When a similar test was made with different Windows Mobile equipment, everything went fine and no hangs were felt.

Table 5-5: Observed Failure Modes in Scenario A.

Field	#Tests & Results	CTS	Ack	CF-End	CF-End CF-Ack	Data	Assoc. Resp	Reass. Resp	Prob. Resp	Beacon	Disass.	Auth.	Deatuth.
Protocol* Version	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
To/From* DS	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
More Flags*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Retry*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Power Management*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
More Data*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
WEP*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Order*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
Duration	3500	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	F1	F1	F1
RA/Addr1	8	F1	F1	F1	F1	F1							
TA/Addr2	8					F1							
DA	8						F1	F1	7x F1C	7x F1C	F1	F1	F1
SA	8						F1	F1	F1A	F1A	F1	F1	F1
AID	15						F1	F1					
BSS ID	8			F1	F1		F1	F1	F1	F1	F1	F1	F1
Addr3	8					F1							
Sequence Control	10					F1	F1	F1	F1	F1	F1	F1	F1
Addr4	7					F1							
Frame Body	7					F1							
Timestamp	6								F1A	F1A		F1	
Beacon** Interval	2700								F1A	F1A		F1	
Capabilities**	2050						F1	F1	F1	F1		F1	
SSID**	1275						F1	F1	32x F1B	32x F1B	F1	F1	F1
Supported** Rates	256						F1	F1	F1A	F1A	F1	F1	F1
FH** Parameter	256						F1	F1	F1A	F1A	F1	F1	F1
DS** Parameter	256						F1	F1	F1A	F1A	F1	F1	F1
CF** Parameter	256						F1	F1	F1A	F1A	F1	F1	F1
IBSS** Parameter	256						F1	F1	F1A	F1A	F1	F1	F1
TIM**	256						F1	F1	F1	1X F6	F1	F1	F1
Reason Code	15										F1		F1
Status Code	5						F1	F1				F1	
Auth. Algorithm Nbr.	5											F1	
Auth. Trans. Nbr.	5											F1	
Other TLV**	1255						F1	F1	F1	F1	F1	F1	F1

*Frame Control; **Tag Length Value

This probably means that the flaw is in HP iPAQ DD. Even so, the vulnerability is critical from an availability standpoint because exploitation is simple (e.g., since Beacon frames are processed in all states, a hacker would only need to walk around with a malicious AP to hang all vulnerable devices in a surrounding area).

The Probe Response failure modes were identical to the Beacon frame, with the exception of the TIM field where no OS hangs were seen.

Failure Modes in Scenario B

To perform the experiments corresponding to the scenario B, the SUT was associated and authenticated to the Real AP using no encryption protocol. The results are shown in Table 5-6. The outcomes for the Beacon and Probe Response frames are equivalent to those obtained in scenario A, which is not surprising, as the process of detecting APs while connected to another AP remains the same.

Fuzzing Disassociation and Deauthentication frames confirmed a known problem with the Wi-Fi protocol. Since the various fields of the frame are not cryptographically protected with some authentication data (e.g., a message authentication code), a rogue AP can transmit Disassociation and Deauthentication frames and cause the Wi-Fi communication to be disrupted (i.e., the Wi-Fi protocol is vulnerable to a Denial of Service (DoS) attack). This can happen if the Destination Address (DA) is equal to the address of the associated STA or the broadcast address. Nevertheless, we found out that several checks are made before accepting the frames, making the attack harder to execute. Several flags of the frame control part of the packet are verified (To/From DS, More Flags, Retry, Power Management, More Data, WEP and Order), reducing significantly the combinations that break the communication.

We also discovered that, whenever the SUT became disassociated and got associated after terminating the attack, the Traffic Generator could not recover the TCP-IP communication. This aspect reveals that some implementation problems may exist in the TCP-IP stack. Contrarily, whenever the SUT become deauthenticate and got authenticated at the end of the attack, the Traffic Generator always recovered the TCP-IP communication. This shows that the DoS attacks performed with Dissassociation frames can be more harmful than the ones made with Deauthentication frames.

Table 5-6: Observed Failure Modes in Scenario B and C.

Field	#Tests & Results	CTS	Ack	CF-End	CF-End CF-Ack	Data	Assoc. Resp	Reass. Resp	Prob. Resp	Beacon	Disass.	Auth.	Deatuth.
Protocol* Version	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
To/From* DS	4	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	3x F3	F1	3x F4
More Flags*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Retry*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Power Management*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
More Data*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
WEP*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Order*	2	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	1x F3	F1	1x F4
Duration	3500	F1	F1	F1	F1	F1	F1	F1	F1A	F1A	3500x F3	F1	3500x F4
RA/Addr1	8	F1	F1	F1	F1	F1							
TA/Addr2	8					F1							
DA	8						F1	F1	7x F1C	7x F1C	2x F3	F1	2x F4
SA	8						F1	F1	F1A	F1A	1x F3	F1	1x F4
AID	15						F1	F1					
BSS ID	8			F1	F1		F1	F1	F1	F1	1x F3	F1	1x F4
Addr3	8					F1							
Sequence Control	10					F1	F1	F1	F1	F1	1x F3	F1	1x F4
Addr4	7					F1							
Frame Body	7					F1							
Timestamp	6								F1A	F1A		F1	
Beacon** Interval	2700								F1A	F1A		F1	
Capabilities**	2050						F1	F1	F1A	F1A		F1	
SSID**	1275						F1	F1	32x F1B	32x F1B	1275x F3	F1	1275x F4
Supported** Rates	256						F1	F1	F1A	F1A	256x F3	F1	256x F4
FH** Parameter	256						F1	F1	F1A	F1A	256x F3	F1	256x F4
DS** Parameter	256						F1	F1	F1A	F1A	256x F3	F1	256x F4
CF** Parameter	256						F1	F1	F1A	F1A	256x F3	F1	256x F4
IBSS** Parameter	256						F1	F1	F1A	F1A	256x F3	F1	256x F4
TIM**	256						F1	F1	F1	1x F6	256x F3	F1	256x F4
Reason Code	15										15x F3		15x F4
Status Code	5						F1	F1				F1	
Auth. Algorithm Nbr.	5											F1	
Auth. Trans. Nbr.	5											F1	
Other TLV**	1255						F1	F1	F1	F1	1255x F3	F1	1255x F4

*Frame Control; **Tag Length Value

Failure Modes in Scenario C

In scenario C, the test campaign was performed with the SUT associated and authenticated to the Real AP using shared key mode encryption protocol. The results observed in scenario C were equal to the ones obtained in the scenario B.

5.8 Summary

The Wdev-Fuzzer tool is a fuzzer that targets DDs of communication protocols. The proposed architecture is quite generic, allowing a detailed description of the protocol's messages. Therefore, the generated attacks are very effective at discovering new vulnerabilities and at verifying known issues. Additionally, the tool can also help to perform some of the tasks of conformance testing, by detecting misbehaviours of the DD's implementation with respect to the specification of the protocols.

The presented version of the tool was utilized to evaluate a Wi-Fi DD of a smart phone running Windows Mobile 5. The results demonstrated that in most cases, Windows was able to handle correctly the malicious frames. They also showed that Wdev-Fuzzer can be successfully applied to reproduce denial of service attacks using Disassociation and Deauthentication frames. The tool revealed that there might be a problem in the implementation of the TCP-IP stack, uncovered by the use of disassociation frames when the SUT was associated and authenticated with an AP. Finally, it discovered a previously unknown vulnerability that causes OS hangs, using the TIM element in the Beacon frame.

CHAPTER 6 INTERCEPT

There is a significant difference between being able to trigger an error and locate the vulnerability behind the error. Locating the flaw requires access to the system under test in such a way that it is possible to pinpoint the part of the code that is responsible for the observed behaviour. In the case of Windows DDs (WDD) this is a challenge. In most times, it is impossible for independent researchers to have access to the source code of the DD, making it hard to understand the reasons behind a faulty behaviour.

This chapter describes the Intercept tool that focus on DD involved with communications that can instrument WDD by logging data about the interactions with the OS. It operates without access to the driver's source code and with no changes to the driver’s binary file. As its name indicates, the tool intercepts all function calls between the DD and the OS, ensuring that various information can be collected, such as the name of the functions that are invoked, their parameters and return values, and the content of particular areas of memory. Although simple in concept, it enables the users to expose a DD behaviour and data structures, which provide a practical approach towards its understanding.

Intercept can be used as a building block of other tools by providing the contents of packets and the context of their arrival/departure. For this purpose, Intercept can log the network traffic information in the format used by Libpcap [98], which can then be analysed by popular tools such as WireShark [99]. Intercept can be very helpful

in debugging processes since it gives a higher level vision of what is happening between the OS and the driver, and at the same time offering information on the parameter contents and address locations. Combined with debugging tools from Microsoft, such as WinDbg [101], this data is useful to reduce the time for locating functions, OS resources and global variables.

Intercept logs information about the interactions between the OS core and the DD under test (DUT). The data is collected during the whole period of execution, starting when the driver is loaded and ending when it is uninstalled. It includes among others, the list of functions that are used, the order by which they are called, and parameter and return values. This information is quite comprehensive, and it helps not only to understand the driver-OS interactions, but also to realize how drivers deal with the hardware in terms of programming and access to specific storage areas.

Intercept uses an approach to instrument DDs in Windows that requires no changes to the binary code. It resorts to a proxy DD that points all imported functions from a driver to its own interception layer. Call-back functions registered by the driver are also captured and directed to the interception layer. No extra code needs to be developed for normal operation - a complete log is generated describing how the driver behaves as a result of the experiments. However, extensibility is achieved by changing the actions performed by the interception layer, allowing more complex operations to be carried out.

6.1 Intercept Architecture

The architecture of Intercept is represented in Figure 6-1. It can be divided in two main components: The Intercept Windows DD (IWDD) and the Intercept User Interface (IUI). The first is a Windows driver that provides all the necessary functions to load, execute and intercept the DUT. The second is an application that allows users to setup the interception process and control the IWDD activity.

The components of IWDD are the following. The Controller provides an interface for the IUI application to control the behaviour of the IWDD, allowing for instance the definition of the level of detail of logging and the selection of which functions should be logged.

The Loader & Connector (LC) is responsible for loading the "`DUT.sys`" file into the memory space of IWDD. It also links all functions that the DUT calls from external modules to the functions offered by the Interception layer.

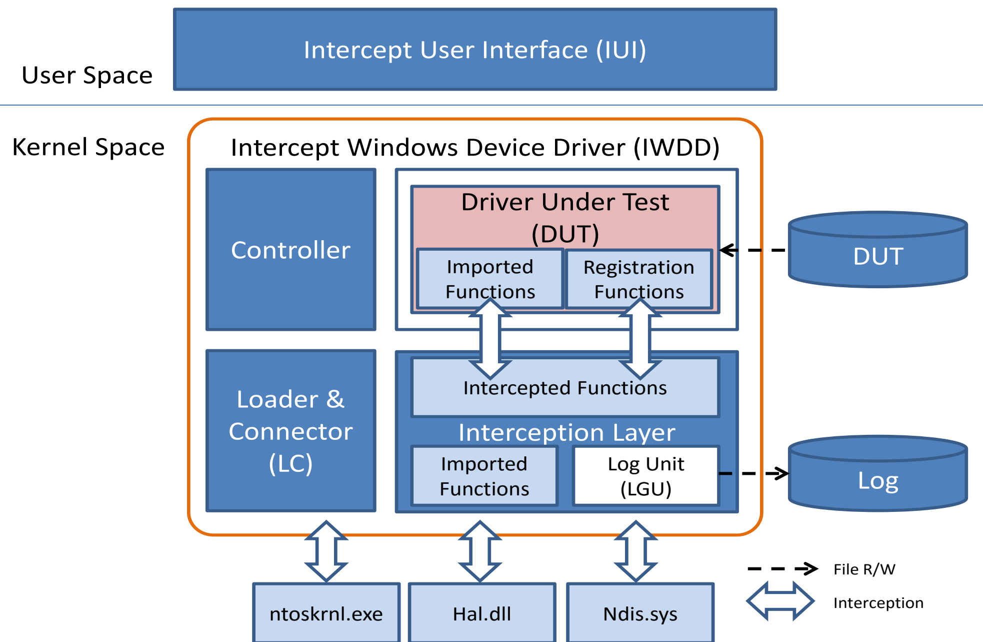


Figure 6-1: Intercept architecture.

The Interception Layer provides the environment for the DUT to run, and intercepts all calls performed by the OS to the DUT and the other way around. The Log Unit (LGU) receives the log entries from the Interception layer and saves them to a file. This is performed in a separate task to decouple the write delays from the remaining processing, and therefore increase the system performance.

6.2 Using Intercept

Intercept is installed by replacing in the system the DUT with its own driver (the IWDD). When the OS attempts to load the DUT, in fact it ends up loading IWDD. Later on, IWDD brings to memory the DUT for execution. Setting up the interception of a DUT involves the following steps:

1. The user indicates the DUT of interest through the IUI interface, where a list of devices present in the OS is displayed;
2. The IUI locates the `DUT.inf` and `DUT.sys` files, and makes a copy of them to a predefined folder. A copy of the `IWDD.sys` file is also placed in the same folder;
3. The IUI replaces in the `DUT.inf` file all references to `DUT.sys` with `IWDD.sys`. The IUI also removes references to the security catalogue, since IWDD is not currently digitally signed. This way, when the OS interprets the `DUT.inf` file, it will install `IWDD.sys` instead;

4. The Windows Device Manager (WDM) is used to uninstall the `DUT.sys`, and then it is asked to check for new hardware, to detect that there is a device without a driver. At that time, the location of the predefined folder is provided, and Windows interprets the modified `DUT.inf` file. Since there is a match with the hardware identification of the device, it proceeds to load the `IWDD.sys` file.

After loading `IWDD.sys`, the following sequence of actions occurs:

1. The WDM calls the `DriverEntry(DriverObject *drvObj, PUNICODE_STRING RegPath)` function of IWDD, so that it can initialize and register the call-back functions. Parameter `*drvObj` is a complex structure where some of the exported call-back functions can be registered. Parameter `RegPath` is the path of the Windows Register location where the driver should store information. Since the DD functionality is to be provided by the original DUT implementation, at this stage the control is given to the LC unit to load the DUT's code;

2. The LC unit interprets the `DUT.sys` file contents, relocates the addresses, and goes through the table of imported functions to link them to the Interception layer. Technically this is achieved by having in the Interception layer a table containing entries with a 'name' and an 'address' for each function. The 'name' is the Windows function name that can be found in the imported table of the DUT and the 'address' is a pointer to the code of the function. The 'address' of the function in the Interception layer is placed in the imported function table of the DUT's. In the end, all imported functions of the DUT point to functions in the IWDD.

3. Next, the `DUT.sys` binary is merged and linked to the IWDD. The LC unit also finds the address of the DUT's `DriverEntry()`, which is then executed. As with any other driver, the DUT has to perform all initializations within this function, including running `NdisMRegisterMiniportDriver()` to register its exported functions to handle packets. However, since the DUT's imported functions were substituted by IWDD functions, a call to `NdisMRegisterMiniportDriver()` in fact corresponds to a call to `_IWDD_NdisMRegisterMiniportDriver()`[2]. In the particular case of this function, the DUT gives as parameters the call-back functions to be registered in the NDIS library. In the Interception layer, the implementation of this function swaps the function addresses with its own functions, making the interception effective also for functions that will be called by the OS to the DUT.

---

[2] The prefix _IWDD_ is used to identify a function provided by the IWDD.

4. When the DUT's `DriverEntry()` finishes, it returns a `drvObj` parameter containing potentially also some pointers to call-back functions. Therefore, before giving control back to the OS, IWDD replaces all call-back entries in `drvObj` with its own intercept functions, which in turn will call the DUT's routines. This way this type of call-back function is also intercepted.

6.3 Tracing the Execution of the DUT

The DUT starts to operate normally, but every call performed by the OS to the DUT, and vice versa, is intercepted. The Interception layer traces all execution of the DUT, recording information about which and when functions are called, what parameter values are passed, which return values are produced and when the function exits. The log uses a plain text format and data is recorded to a file.

All functions implemented in the Interception layer make use of routines `_IWDD_DbgPrint()` and `_IWDD_Dump(char *addr, long size)`. The first works like the C language `printf()` function, and is used to write formatted data to the log file, such as strings and other information types. The second function is used to dump into the log file the contents of memory of a certain range of bytes starting at a given memory addresses. Together, these two functions can give a clear insight of the DUT's and OS's interaction.

Typically, the Interception layer creates a log entry both when entering and leaving a function. Whenever input parameter values are involved, they are also logged before calling the intended function, either in the DUT's code or in the OS. Output parameters and return values are saved before the function ends execution. Complex structures, such as `NetBuffers`, `NetBufferLists` or `MDLs`, are decomposed by specific routines so that the values in each field of the structure can be stored.

The interception of functions and the trace of its related information is a time-consuming activity that may interfere with the DUT and the overall system performance. To reduce overheads, the storage process is handled by a separate thread. During the IWDD start-up process, the LGU unit creates a queue and a dedicated thread (`DThread`), whose task is to take elements from the queue and write them into the log file. The queue acts as a buffer to adapt to the various speeds at which information is produced and consumed by the thread. The access to the queue is protected by a lock mechanism to avoid race conditions. A call to `_IWDD_DbgPrint()` or `_IWDD_Dump()` copies the contents of the memory to the queue, and signals the thread to wake up and store the information.

In the standard mode of operation, the log file is created when the thread is initiated. Each time the thread awakes, the data is removed from the queue and written to the file. When the file reaches a pre-determined value, it is closed and a new one is created. However, in case of a crash, the information in cache can be lost. To cope with this situation, the thread can also be configured to open, write synchronously and close the file each time it consumes data from the queue. However, this comes at the expense of a higher overhead.

6.4 Experimental Results

The objective of the experiments is twofold. First, we want to get some insights into the overheads introduced by Intercept, while a DD executes a common network task - a file transfer by FTP. Second, we want to show some of the usage scenarios of the tool, such as determining which functions are imported by the drivers and what interactions occur while a driver runs.

Test environment

The experiments were performed with three standard drivers, implementing different network protocols, namely Ethernet, Wi-Fi and Bluetooth. Table 6-1 summarizes the installation files for each DUT.

The corresponding hardware devices were connected to a Toshiba Satellite A200-263 Laptop computer. The Ethernet and Wi-Fi cards were built-in into the computer, while the Bluetooth device was a SWEEX Micro Class II Bluetooth peripheral [100] linked by USB. In the tests, we have used Intercept both with Windows Vista and Windows 8.

Table 6-1: Device drivers under test.

Driver Type	Info File	Binary file
Ethernet	netrtx32.inf	rtlh.sys
Wi-Fi	netathr.inf	atrh.sys
Bluetooth	netbt.inf	btnetdrv.sys

The overhead experiments were based on the transmission of a file through FTP. The FTP server runs in an HP 6730b computer. The FTP client was the Microsoft FTP client application, which was executed in the laptop together with Intercept. Different network connections were established depending on the DUT in use. For the Ethernet driver an Ethernet network of 100Mbps using a TP-Link 8 port

10/100Mbps switch was setup to connect the two systems. For the Wi-Fi and Bluetooth drivers an ad-hoc connection was established.

Overhead of Intercept

To evaluate the overheads introduced by Intercept, we have run a set of experiments consisting on the transfer of a file of 853548 byte length between a FTP server and a client. Any file could have been used for the transfer. We selected this file because it was the first log produced by Intercept during the experiments.

For each driver five FTP transfers were performed, and the average results are presented in the tables. Table 6-2 summarizes the results for the execution time and transfer speeds. Column “Driver ID” represents the DUT, either in Windows Vista (xx_Vista) or in Windows 8 (xx_Win8). The columns under the label “Intercept off” display the average transfer time and average speed when the Intercept tool is not installed in the client system. The columns under label “Intercept on” correspond to the case when the Intercept tool is being used.

The results between Intercept off and on show a performance degradation, which was expected as Intercept records all the activity of the drivers, and performs tasks such as decoding parameter structures and return values of all functions. Nevertheless, these overheads are relatively small: between 2% and 7% for the Ethernet driver, 2% to 3% for the Bluetooth driver and 14% to 15% for the Wi-Fi driver. These observations were more or less expected since the Wi-Fi drivers have more imported functions, are longer in size and require more processing when compared with the other drivers. The same Bluetooth driver was used in both OS which can explain the similarity of the degradation.

Table 6-2: Average file transfer time and speed values.

Driver ID	FTP Transfer				
	Intercept Off (average)		Intercept On (average)		Time overhead
	Time*	Speed**	Time*	Speed**	
Eth_Vista	0,198	6238	0,202	6204	2%
Eth_Win8	0,136	6503	0,146	5963	7%
Wi_Fi Vista	9,300	97	10,650	84	15%
WiFi_Win8	0,276	3076	0,314	2872	14%
Bth_Vista	5,890	145	6,012	142	2%
Bth_Win8	5,612	152	5,760	148	3%

Note: *time in seconds, **speed in Kbytes/second

The differences between the overheads on the Ethernet and Wi-Fi networks can be related to changes in the drivers, since we have used the standard drivers that came with the Windows installation.

During the experiments, we saw that for each transmitted byte, Intercept generated between 9 to 23Kbytes of data. Not surprisingly the Wi-Fi driver was the one that generated a higher amount of data, which can be interpreted as a synonymous of increased complexity.

Understanding how drivers are initialized

Although there is plenty literature about Windows DDs (see for instance[157][158][159][160][161][162]) and source code examples (see for instance [163][164][165][166]), programming this type of modules is not an easy task. Intercept contributes to understanding the DD behaviour since the moment the DD is loaded and initialized. As an example, Figure 6-2, shows the moment when the DD registers its call back functions on Windows using function `NdisMRegisterMiniportDriver`.

Obtaining this type of information allows one to understand some of the DDs characteristics (such as versioning information) and map the location of the DD's call-back functions and objects, which can be useful during debugging or reverse engineering processes.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-     CALLING DUT ENTRY POINT    -+-+-+-+-+-+-+-+-+-+-+-+-
*****************************************
ENTER - _My_NdisMRegisterMiniportDriver
DriverObject..................: 0x8aeff480
RegistryPath..................: 0x880c7000
MiniportDriverContext.........: 0x00000000
MiniportDriverCharacteristics.: 0x8c36b618
NdisMiniportDriverHandle......: 0x89acaeac
*NdisMiniportDriverHandle.....: 0x00000000
--------- MiniportDriverCharacteristics ---------
Header Type...................: 0x0000008a
Header Size...................: 0x00000048
Header Revision...............: 0x00000001
MajorNdisVersion..............: 0x00000006
MinorNdisVersion..............: 0x00000000
MajorDriverVersion............: 0x00000001
MinorDriverVersion............: 0x00000009
Flags.........................: 0x00000000

Save Old InitializeHandlerEx
OldMPInitializeEx...........................: 0x89a1c400
NewMPInitializeEx...........................: 0x8f811ac0
Save Old HaltHandlerEx
OldMPHalt...................................: 0x89a1d790
NewMPHalt...................................: 0x8f811b60
```

Figure 6-2: Drive initialization – Call to NdisMRegisterMiniportDriver.

Understanding how drivers interact with the hardware

Intercept can also help to understand how specific hardware interactions are performed. The NDIS Library provides a set of I/O functions that a miniport driver calls to access I/O ports. These calls provide a standard portable interface that supports the various operating environments for NDIS drivers. For instance, functions are offered for mapping ports, for claiming I/O resources, and for reading from and writing to the mapped and unmapped I/O ports. Taking the Wi-Fi driver as an example, one can use Intercept to learn how the hardware initialization process happens. It starts when the OS invokes the drivers' call-back function `MPInitializeEx` (see Figure 6-3).

The OS passes several parameters to this function. One of them is the `MiniportAdapterHandle` so that whenever there is the need for the driver to call for some function, the OS is able to know which hardware the driver is referencing to (in this case, the reference is `0x8b34a438`). All subsequent functions related with this driver will use this reference.

```
ENTER - NewMPInitializeEx
MiniportAdapterHandle.....: 0x8b34a438
MiniportDriverContext.....: 0x00000000
MiniportInitParameters....: 0x8c36b6c0
Header.Revision..................................: 0x00000001
Header.Size......................................: 0x00000028
Header.Type......................................: 0x00000081
Flags............................................: 0x00000000
IMDeviceInstanceContext..........................: 0x00000000
MiniportAddDeviceContext.........................: 0x00000000
IfIndex..........................................: 0x0000003f
NetLuid..........................................: 0x00000000
NetLuid.Info.....................................: 0x00000000
NetLuid.Value....................................: 0x00000000
AllocatedResources...............................: 0x8815ccd4
AllocatedResources->Version......................: 0x00000001
AllocatedResources->Revision.....................: 0x00000001
AllocatedResources->Count........................: 0x00000003
AllocatedResources->PartialDescriptors[00000000].Type.........................: 0x00000003
AllocatedResources->PartialDescriptors[00000000].ShareDisposition.............: 0x00000001
AllocatedResources->PartialDescriptors[00000000].Flags........................: 0x00000080
CmResourceTypeMemory
AllocatedResources->PartialDescriptors[00000000].u.Memory.Start...............: 0xd4000000
AllocatedResources->PartialDescriptors[00000000].u.Memory.Length..............: 0x00010000
```

Figure 6-3: Call to MPInitializeEx to initialize the hardware (excerpt).

Another parameter is the resources allocated for the hardware. This allocation was performed automatically by the system according to the PCI standard, which releases the programmers from doing it. However, the driver only gets to know it when this function is called. In this example, some of resources assigned to the Wi-Fi hardware were: Memory start: `0xd4000000` and Memory length: `0x00010000`.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- NdisMGetBusData +-+-+-+--+-+-+-+-+-+-+-+-+-+-+-
ENTER - NdisMGetBusData
MiniportAdapterHandle.......: 0x8b34a438
WhichSpace..................: 0x00000000
Offset......................: 0x00000000
Buffer......................: 0x8c36b500
Length......................: 0x00000040
0x8c36b500  8c 16 1c 00 46 01 10 00 01 00 00 02 10 00 00 00 ....F...........
0x8c36b510  04 00 00 d4 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x8c36b520  00 00 00 00 00 00 00 00 01 50 00 00 4f 14 28 71 .........P..O.(q
0x8c36b530  00 00 00 00 40 00 00 00 00 00 00 00 12 01 00 00 ....@...........
EXIT - NdisMGetBusData



ENTER - _My_NdisMSetBusData
MiniportAdapterHandle.......: 0x8b34a438
WhichSpace......: 0x00000000
OffSet..........: 0x00000004
Buffer..........: 0x8c36b4b8
Length..........: 0x00000002
0x8c36b4b8  46 03                                           F.
res.......: 0x00000002
EXIT - NdisMSetBusData
```

Figure 6-4: Call to NdisGetBusData / SetBusData.

Other examples of interaction with the hardware can give insights of specific register and available ports as is the case of Figure 6-4 that shows the moment that the DD receives data from the PCI bus (using `NdisMGetBusData`) and programs the device by writing some data using `NdisMSetBusData`.

Inspecting data packets

Intercept can also be employed when particular information needs to be collected. As an example, we wanted to find out what data is returned by the FTP server after the client connects. Figure 6-5 shows a call performed by the DUT to the OS notifying NDIS that a new frame has just arrived. In this case, it is possible to observe the banner received from the FTP server, i.e., `220-Welcome to Cerberus FTP Server`.

This type of inspection is possible because Intercept knows the kind of structures involved in each OS function and is able to decompose them. The interpretation and decomposition of complex structures (as data packets) can be extended in Intercept to cope with evolutions of the OS and protocols. A file with the description of the structures and the type of the elements that compose the structure is all that is needed to change the behaviour of the interpreter.

```
+-+-+-+-+-+-+-+-+-+-+-+-+ NdisMIndicateReceiveNetBufferLists +--+-+-+-+-+-+-+-+-+-+-+-+-

ENTER - NdisMIndicateReceiveNetBufferLists
MiniportAdapterHandle..:0x8b34a438
NetBufferLists.........:0x884a6948
PortNumber.............:0x00000000
NumberOfNetBufferLists.:0x00000001
ReceiveFlags...........:0x00000001
NDIS_RECEIVE_FLAGS_DISPATCH_LEVEL
NetBuf..................: 0x884a6948
Dumping mdl.....: 0x880703e0
0x8857b000  08 00 2c 00 00 1b 9e a8 f4 60 e0 b9 a5 5c b8 c7 ..,......`...\..
0x8857b010  ee 15 2c c8 f4 60 30 00 aa aa 03 00 00 00 08 00 ..,..`0.........
0x8857b020  45 00 00 a6 63 b7 40 00 80 06 82 f1 c0 a8 c9 2c E...c.@........,
0x8857b030  c0 a8 c9 2b 00 15 c0 14 95 f1 c5 43 7f ee 95 15 ...+.......C....
0x8857b040  50 18 00 44 72 2a 00 00 32 32 30 2d 43 65 72 62 P..Dr*..220-Cerb
0x8857b050  65 72 75 73 20 46 54 50 20 53 65 72 76 65 72 20 erus FTP Server
0x8857b060  50 65 72 73 6f 6e 61 6c 20 45 64 69 74 69 6f 6e Personal Edition
0x8857b070  0d 0a 32 32 30 2d 55 4e 52 45 47 49 53 54 45 52 ..220-UNREGISTER
0x8857b080  45 44 0d 0a 32 32 30 2d 57 65 6c 63 6f 6d 65 20 ED..220-Welcome
0x8857b090  74 6f 20 43 65 72 62 65 72 75 73 20 46 54 50 20 to Cerberus FTP
0x8857b0a0  53 65 72 76 65 72 0d 0a 32 32 30 20 43 72 65 61 Server..220 Crea
0x8857b0b0  74 65 64 20 62 79 20 47 72 61 6e 74 20 41 76 65 ted by Grant Ave
0x8857b0c0  72 65 74 74 0d 0a                               rett..
EXIT - NdisMIndicateReceiveNetBufferLists
```

Figure 6-5: Looking in detail at a particular packet (excerpt).

Understanding complex interactions with the OS

Intercept can be used to comprehend how certain complex operations are performed by the driver. For example, in Windows, a driver can remain installed but disabled. By analysing the log produced by Intercept during the disabling process, it is possible to observe that the OS first calls the drivers' `MiniportPause` to stop the flow of data through the device. Second, the OS calls `MiniportHalt` to obtain the resources that were being utilized. Both these two functions were registered during the initialization process, at the time using the `NdisMRegisterMiniportDriver` function. Finally, the OS calls the `Unload` function to notify the driver that is about to be unload. The `Unload` function was also registered by the driver in the OS when the `DriverEntry` routine returned, by setting the address of this function in the `DriverUnload` field of the `Driver_Object` structure. As soon as the `Unload` function starts it is possible to observe in the log that the driver calls the `MPDriverUnload` call-back function (see Figure 6-6). When this function ends the unload process ends and the driver is disabled.

Another example corresponds to uninstalling the driver. With the information logged by Intercept, it was found that there is no difference between disabling and uninstalling a driver, except from the fact that uninstalling the driver removes it from the system.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ Unloading +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
ENTER - NdisFreeMemoryVirtual Address........: 0x8817c328
Length.................: 0x00000000
Memory Flags...........: 0x00000000
EXIT - NdisFreeMemory

EXIT - NewMPHalt

DriverUnload Enter
Calling DriverUnload of the driver...
ENTER - MPDriverUnload
ENTER - _My_NdisMDeregisterMiniportDriver
NdisMiniportDriverHandle.........: 0x88107dc8
EXIT - NdisMDeregisterMiniportDriver

ENTER - NdisFreeMemoryVirtual Address........: 0x89866000
Length.................: 0x00000000
Memory Flags...........: 0x00000000
EXIT - NdisFreeMemory
```

Figure 6-6: DD disabling process (excerpt).

Determining resource leakage

The detailed information stored by Intercept in the log also helps to determine if all resources allocated by the driver are returned to the OS core. This can assist for instance to detect drivers with bugs. Table 6-3 represents the list of five resources allocation functions utilized by the Wi-Fi driver and Table 6-4 represents the list of five corresponding de-allocation functions utilized by the same driver. As it is possible to observe, there is a match between the number of resource allocations and releases which shows no resource leakage during the DD execution.

Table 6-3: Statistics of resource allocation/deallocation.

Function	Number of Calls
_IWDD_NdisAllocateIoWorkItem	1158
_IWDD_NdisMAllocateNetBufferSGList	1041
_IWDD_NdisMAllocateSharedMemory	803
_IWDD_NdisAllocateNetBuffer	256
_IWDD_NdisAllocateNetBufferList	256

Table 6-4: Statistics of resource allocation/deallocation.

Function	Number of Calls
_IWDD_NdisFreeIoWorkItem	1158
_IWDD_NdisMFreeNetBufferSGList	1041
_IWDD_NdisMFreeSharedMemory	803
_IWDD_NdisFreeNetBuffer	256
_IWDD_NdisFreeNetBufferList	256

Understanding the dynamics of function calls

The dynamics of function calls during a driver's execution is determined by its work load. Intercept can support various kinds of profiling analysis about the usage of functions by a certain DD under a specific load. For example, in our FTP transfer scenario, Table 6-5 represents the top 5 most called functions by each DUT from installation and until deactivation (in Windows Vista).

Table 6-5: Top 5 most used functions by each driver.

Function	Eth_Vista	WiFi_Vista	Bth_Vista
NdisMSynchronizeWithInterruptEx	-	69301	-
InterruptHandler	880	33931	-
MiniportInterruptDpc	-	32774	-
NdisAcquireReadWriteLock	-	6345	-
NdisReleaseReadWriteLock	-	6345	-
NdisMIndicateReceiveNetBufferLists	-	-	1032
NdisAllocateMdl	1096	-	-
NdisFreeMdl	1096	-	-
NdisAllocateNetBufferAndNetBufferList	1024	-	-
NdisFreeNetBufferList	1024	-	-
NdisAllocateMemoryWithTagPriority	-	-	520
NdisFreeMemory	-	-	520
MPSendNetBufferLists	-	-	503
NdisMSendNetBufferListsComplete	-	-	503

Based on the number of function calls it becomes clear that the Wi-Fi driver is the one that shows more activity in the system. Focusing on the top 3 functions from this driver, the `NdisMSynchronizeWithInterruptEx` is the most used function.

Drivers must call this function whenever two threads share resources that can be accessed at the same time. On a uniprocessor computer, if one driver function is accessing a shared resource and is interrupted, to allow the execution of another function that runs at a higher priority, the shared resource must be protected to prevent race conditions. On an SMP computer, two threads could be running simultaneously on different processors and attempting to modify the same data. Such accesses must be synchronized.

`InterruptHandler` is the second most executed function. This function runs whenever the hardware interrupts the system execution to notify that attention is required. From the 33931 interrupts, 32774 calls were deferred for later execution with `MiniportInterruptDpc`. By inspecting the remaining functions used by the Wi-Fi driver, which are lock related, it becomes evident that the driver is relying heavily on multithreading and synchronization operations.

Several other metrics can be obtained with Intercept, such as the minimum, average and maximum usage of each individual resource, DMA transfers, restarts, pauses, most used sections of the code, to name only a few.

Using Intercept as a Testing Tool

Due to the detailed logs provided by Intercept, a tester can fully understand the driver's dynamics, and thus plan and design tests that target specific and elaborated conditions. During the call to a function Intercept can identify the presence of specific conditions specified by the tester to interfere with parameters and return values.

For instance, the Wi-Fi driver in Windows 8 calls the `NdisMMapIoSpace` during the initialization. This function maps a given bus-relative "physical" range of device RAM. When successful, this function returns `NDIS_STATUS_SUCCESS` and the value of the output parameter `VirtualAddress` contains the start of the memory map. Other outcomes are exceptions that should be handled quietly.

We have performed a series of experiments when the DD called `NdisMMapIoSpace` during the initialization. Four test scenarios where planned by returning to the DD exceptional values (as described in Microsoft documentation) `NDIS_STATUS_RESOURCE_CONFLICT`, `NDIS_STATUS_RESOURCES`, `NDIS_STATUS_FAILURE` and one unspecified value (`NDIS_STATUS_FAILURE+1`), while maintaining the `VirtualAddress` equal to `NULL`. The DUT handled correctly the tests and ended quietly, and appropriately deallocated all resources, as confirmed by the Intercept logs.

Four additional test scenarios were performed with the same return values but assigning a specific value to `VirtualAddress`. These tests all resulted in a crash of the system with the DUT being the culprit. It was concluded that the driver is using the value of `VirtualAddress` before checking the return value, which is worrisome in case Windows does not clear the `VirtualAddress` field.

6.5 Summary

This chapter presents Intercept, a tool that instruments WDD by logging the driver interactions with the OS at function level. It uses an approach where the WDD binary is in full control and the execution traced to a file recording all function calls, parameter and return values. The trace is directly generated in clear text with all the involved data structures.

An experiment with three network drivers was used to demonstrate some of the instrumentation capabilities of Intercept. The performance of the tool was also evaluated in a FTP file transfer scenario, and the observed overheads were small given the amount of information that is logged, all below 15%.

As is, Intercept gives a clear picture of the dynamics of the driver, which can help in debugging and reverse engineering processes with low performance degradation.

Intercept is also a building block for a testing tool. Results show the ability to identify bugs in drivers, by executing tests based on the knowledge obtained from the driver's dynamics.

CHAPTER 7 SUPERVISED EMULATION ANALYSYS

Experimentally testing a DD typically requires a target host setup composed by a computer running the full OS installation and the hardware driven by the DD under test. To manage the experiments, it is usually required a second system that controls the tests and monitor the results. This is necessary because a bug in the DD can corrupt the execution environment of the experiments as well as the collection of the results. The delays introduced by the need to restart the host system and setup the initial conditions can slow the testing campaign. One way to speed up the restauration process and avoid some of the effort required to manage all the restart actions is to use virtual machine execution. In this case, the virtual machine contains a snapshot image of the system under test which, in the case of a crash or hang, can allow the system to reinitiate from a previous saved starting point (see for instance [53][54]). However, the required setup is still there. One needs the full OS installation, ensuring that the DD of interest is loaded and that the appropriate workload is produced. Moreover, it is required that the hardware driven by the DD is present. To determine the root cause of a problem, typically further analysis is needed, most of the times relying on the ability of the OS to locate the origins of the problem, which sometimes cannot be performed adequately (see for instance [92]).

In this chapter we define the Supervised Emulation Analysis methodology that supports the identification and location of errors in DDs without the need of the source code and the hardware component. Since testing is carried out with the

binary of the DD, a series of problems related with the dependency of the source code are solved. In addition, inaccuracies introduced by compiler optimizations are detected improving the overall precision of the approach. Another aspect that is addressed is related with the target architecture. Often programmers tend to maintain a single source code for different target architectures by introducing conditional compilation flags that are instantiated for the various deployments. During the compilation process bugs may be introduced, as the final target specificities may not be properly taken into consideration at the time of the driver writing. Finally, the binary of the driver to be installed could have suffered malicious changes after its final compilation, and therefore, testing the DD version that is going to be utilized would allow the discovery of the added weaknesses.

In summary, the motivation for this work originates from the following ideas: i) only use the binary of the DD; ii) no specific hardware is needed and iii) resort to an emulation machine. The combination of these ideas potentiates the implementation of systems that perform DD testing as a service where a distributed and collaborative platform available through the web could allow a faster detection of DD flaws, something especially important for previously unknown code.

7.1 Methodology

In modern systems, user applications cannot communicate directly with the hardware. DDs give support to this task and export interfaces that the OS and the applications can use to access devices creating a uniform layer that abstracts the details of the different hardware.

In the case of the two most popular OS, both, Windows and Linux, share a similar approach in the way that the OS kernel deals with the hardware (this approach is also common to iOS). The similarities found between both OS in the platforms that they run and in the approach taken to address kernel extensions can definitively be used as an argument for the development of a common methodology for the discovery of bugs and vulnerabilities in DDs. However, unlikely to Linux where the majority of the source code is available, on Windows the source code of the DD is usually kept confidential. The Supervised Emulation Analysis is a methodology for the detection and location of flaws in DDs. This methodology is based on the definition of the following elements:

- The assumptions on the DD structure that allows for the methodology to be applied;
- The specification of the DD bug classes that are going to be detected and located;

- The definition of the validators that will be employed to discover the considered bug classes;
- The platform architecture that should be followed by tools implementing this methodology to achieve the desired detection objectives;
- The procedures to be executed to locate flaws in the DDs.

The next sections describe in more detail each of the previous enumerated elements.

7.2 Assumptions on Device Driver Structure

DDs are built according to a DD model determined by the OS internal organization. The driver model (among other things) establishes the internal structure of the DD, defines the interface between the OS and the DD (and vice versa), and the logic sequence of the calls. Additionally, the OS supports the file structure that transports the DD binary code to be loaded into memory for execution.

Device Driver Model

Generically speaking a DD contains several functions that can be grouped in different classes: i) interface, ii) entry point, iii) unloading, iv) internal, v) interrupt and vi) imported. Each of these groups plays a specific role in the work cycle of the driver and understanding them can help to design solutions for testing them.

- **Interface functions.** The interface functions implement services that the DD makes available to the OS. It is included in this category functions such as read, write, power management and IOCTL. These are the functions that the OS interfaces directly to request specific operations.
- **Entry point function.** The driver contains one entry point function responsible for initializing the internal structures of the DD. It is the unique interface function know right after the DD loading. In the majority of OS, the execution of the driver initialization function registers other interface functions made available by the DD to the OS.
- **Unloading function.** This is a special interface function that performs the opposite of the initialization function. It detaches the driver from the hardware, unregisters the DD from the OS and performs all the necessary clean-up, such as returning all the resources that were acquired during the driver working period.

- **Internal functions.** The code of the DD is implemented using a series of internal functions that should simplify the code organization as well as maintainability. These functions cannot be directly interfaced by the OS but are used by some of the interface functions and other internal functions.
- **Interrupt functions.** A driver that deals with hardware typically has associated functions that are called as a result of an external event that triggers them. Typically, the OS already has typified interrupt vectors for each type of device that will be attached to the DD interrupt handlers. Interrupt functions are a special kind of interface functions and are typically registered by the driver initialization routine.
- **Imported functions.** A driver depends on functions typically provided by the kernel. These are the functions provided by the OS that form the API that the DD can use.

Based on the previous information it should be possible to build a system that can interface the DD code and perform the same tasks as the OS. This system could then test the driver through the various functions identified above and be able to locate errors by using test cases that addresses:

- The parameters of the interface;
- The parameters of the interrupt functions and the trigger timings;
- The output return value and output parameters of Imported Functions.

Binary Transport File

The binary image of the DD is normally stored in a file and envelops the binary executable code. Using the appropriate file format and interpreting it according to the specification (e.g., COFF, ELF or EFI) allows the different sections of the DD to be correctly identified. This maps the contents of the file into binary code, and subsequently to memory addresses that will hold the executable code region, data regions, relocation tables and external dependencies of the driver code.

Using this knowledge, it is possible to build a system that processes the DD binary file and performs the tasks that the OS does to prepare the driver for execution.

7.3 Device Driver Flaw Classes

A flaw is a malfunction in a program that makes it to produce incorrect outputs, behave in an undesired way, such as terminate unexpectedly. When bugs are not a consequence of a programming error, they are usually a consequence of design flaws.

Although software can be affected by an enormous quantity of flaws (see for instance the Common Weakness Enumeration (CWE) classes [131] or the Seven Pernicious Kingdoms taxonomy [132]), the typical error classes affecting the DD code is a more restrictive subset. The goal of identifying the bug classes that may affect DDs is to characterize the kinds of flaws that a given tool is able to identify and the instruments that need to be built to detect them.

Flaw classes are intrinsically connected to the underlying design of the execution platform and architecture of the OS. For instance, in a x86 platform running Windows, there are several calling conventions that determine the usage of the stack to pass arguments to functions. In the x64 platform also with Windows, parameters are passed using registers instead. This correlation between the calling convention and the execution platform changes the type of bugs that can affect the target platform where the DD is executed and consequently the type of mechanisms necessary to detect them.

The following sections describe typical flaw classes that commonly affect DDs.

Uninitialized/ Nonvalidated/ Corrupted Pointers

Whenever a pointer is dereferenced, it is retrieved the value contained at the memory address location hold by the pointer. For example, the C language standard defines that a static uninitialized pointer has a `NULL (0x00)` value. If a kernel path attempts to dereference a `NULL` pointer, it will try to access the memory address `0x00`, which likely will result in a halt or hang condition, since the protection mechanism of the platform knows that nothing is mapped there.

`NULL` pointer dereference vulnerabilities are a subset of a larger class of bugs known as uninitialized/ nonvalidated/ corrupted pointer dereference. This category covers all situations in which a pointer is used while its content has been changed, was never properly set or was not correctly validated. This class covers also incorrect sequence of function calls. For instance, many resources can only be used by the DD if they are properly initialized and allocated. Access to memory through pointers without a proper initialization will normally refer to an incorrect memory area. Corrupted pointers can also be a consequence of some other type of bugs,

such as buffer overflows, which change one or more of the bytes where the pointer is stored.

Stack Related Flaws

The kernel stack implementation follows conventions that include the growth direction (from higher addresses to lower addresses, or vice versa), the register that keeps track of its top address, the location where local variables are saved, how parameters are passed, and how a sequence of function calls is linked together.

Kernel stack vulnerabilities are usually the consequence of writing past the boundaries of a stack allocated buffer. This kind of situation can occur as a result of using unsafe C functions, such as `strcpy()` or `sprintf()`, since these functions keep writing to their destination buffer, regardless of its size, until a `0x00` terminating character is found in the source string. An incorrect termination condition in a loop that populates an array is also an example of how such situation can occur. Another example is in the use of one of the safe C functions, such as `strncpy()`, `memcpy()`, or `snprintf()`, but incorrectly calculating the size of the destination buffer.

The stack plays a critical role in the application binary interface and the detection of stack vulnerabilities can be heavily architecture-dependent.

Heap Vulnerabilities

The kernel implements a virtual memory abstraction, creating the illusion of a large and independent virtual address space. The kernel continuously manages space for a large variety of small objects and temporary buffers. The vulnerabilities that can affect the kernel heap are usually a consequence of buffer overflows, triggered by the use of unsafe functions, incorrect loop termination, and incorrect use of safe functions as explained before. The probable outcome of such an overflow is to overwrite some random kernel memory or paging metadata, causing some undesirable behavior.

7.4 Detecting Flaws with Validators

A validator is a mechanism that is called during the DD code execution to perform a check over an intended action. Validators can be defined at the lowest execution level in the platform (such as instruction machine level), at the function interface level or at the end of a sequence of function calls. When a validator is triggered the execution of the code is halted and an error is signalled.

The methodology identifies three different kinds of validator classes:

- **Machine Level Validators (MLV)**: These validators are triggered during the execution of a machine instruction and the objective is to check the parameters involved in the machine instruction. The machine instruction is not executed if the validator returns `false`.
- **Function Level Validators (FLV)**: They are triggered during the execution call from the DD to the OS, therefore embedded in the imported functions code. The implementation of the FLV depends on the type of the called function. Some of these validators may focus on parameter values, while others may be related with the status of a state machine of OS objects.
- **Post Execution Validators (PEV)**: They are triggered after the execution of a sequence of DD interface functions to detect abnormal situations such as, the status of the resources allocated/released or the existence of dormant code.

These types of Validator classes represent different execution levels involved in the analysis of the DD code. While MLV act at the machine instruction level, FLV operate at the interface of DD with the OS. This distinction is necessary, for example: while at the machine level there is nothing wrong in assigning the value `NULL` to a variable and pass the variable value to function `fx`, at the function level the `NULL` value in a handler parameter of function `fx` (that is expecting a valid handler from the OS), may be synonymous of a flaw. Finally, PEV act at the top of the execution level as it depends on the order of calls performed.

Next are several examples of basic validators that should be available to detect the identified flaw classes in the previous section. Tools implementing this methodology should however keep open the possibility to extend these validators.

- **MLV1-Source operand validation**. Checks that the source operand address of an instruction is valid in the context of the operation. Valid source addresses include: i) stack addresses assigned to the function holding the instruction, ii) memory requested by the DD using the imported functions, iii) objects created by the OS Emulator and, iv) hardware location map. The `call` and `jmp` instructions are not covered by this validator.
- **MLV2-Destination operand validation**. Performs the same validations detailed for the source operand validation but applicable to the destination address of the instruction. The `call` and `jmp` instructions are not covered by this validator.

- **MLV3-Call, jmp and ret destination addresses**. The `call`, conditional and unconditional jump and `ret` instructions are subject to a special validation. The destination address of these instructions must fall into the beginning of an internal function of the driver, a jump table located at the DD executable code or into one of the imported functions of the OS.
- **FLVx-Function validators**. Checks inside each of the imported functions to verify the conformity of the parameter values according to the context of the invoked function. Such checks include the parameter type and the allowed interval of values. It is also the responsibility of this type of validators to determine if the prerequisites for executing a function are met. For instance, they should ensure that before calling function B, function A has been called.
- **PEV1-Memory Balance**. Checks, after a determined sequence of interface function calls, the balance of memory allocations, guaranteeing that all allocated memory in function X was freed at function Z.

7.5 Platform Architecture

The objective of defining a platform architecture in the methodology is to identify which components should exist and what are the roles of each of them in achieving the detection goals. Next, we present some of the identified components: i) Execution platform; ii) OS Emulator; iii) Device Emulation and iv) Test Manager.

Execution Platform

The execution platform consists of an emulated environment where the DD code is loaded and executed. The environment emulates the architecture where the DD would run (x86, x86-64, other). The execution of each instruction is subject to the action of validators to ascertain the correctness of the execution.

The emulation ensures that there is no need for the hardware of the platform or the device. The level of independency achieved with an execution platform based on emulation allows to test binary code not originally designed for the target platform, i.e., the execution platform may run in Linux and analyse Windows DD code. Additionally, the stability of the testing platform is not compromised by the tests being performed on the driver code because the detection of the errors is made before the execution takes place.

Since the execution platform is emulated, the discovery of the flaws can be distributed over different systems contributing for gains of efficiency using parallelism.

OS Emulator

DDs require the support of the OS for their execution. The OS Emulator provides an API that allows for the implementation of the driver code. The implementation of this methodology requires that all the functions imported by the DD are available at the execution platform. It is necessary that the output parameters and return values are in control of the Test Manager to allow the generation of particular test conditions.

The OS Emulator is also the primary interface with the DD and mimics the tasks of the OS. This component is in charge of loading the DD and maintaining the data structures that support the driver executions, such as kernel objects. It is also in charge of the calls to the initialization functions and interfaces with all the functions made available from the DD accordingly to the instructions of the Test Manager.

Device Emulation

The role of the Device Emulation is to react to the input/output requests performed by the DD code whenever it interfaces with the hardware, giving appropriate responses such that the execution of the DD code can continue. A DD interacts with the hardware component using two different mechanisms: i) directly through in/out machine code instructions and ii) using OS API functions as intermediate.

When the DD uses in/out instructions, it specifies the address of the device and issues the instruction expecting to read/write some type of information. Similarly, when using an API function as intermediate to the hardware, the involved parameters will transport the data from/to the device using the specified signature of the API function.

Device emulation consists on returning to the driver information to be processed through the interface mechanism (in/out instructions or API function) whose contents and results (successful/unsuccessful access) are controlled by the Test Manager.

Without knowing the details of the hardware it becomes challenging to emulate its behavior. For instance, the DD may look for a particular value in a buffer returned by the device to determine proper initialization. The independency of the hardware is achieved by ensuring that the internal functions of the DD that deal with hardware interfaces have all code paths tested. This guarantees that a code path that expects some kind of device behavior is also tested.

Test Manager

The Test Manager is the component in charge of exercising the DD by conducting the execution of the binary code. It uses a set of test cases to exercise the driver according to a predefined testing strategy.

At the function level, it is the Test Manager that sets the conditions for the tests and interacts with the DD. The Test Manager for instance instructs the entry point function of the driver to be called or the interrupt functions to be processed. It is also the Test Manager that defines what should be the behavior of imported functions when called by the driver (e.g., return values and/or output parameters) and controls the OS Emulator and the Device Emulation.

7.6 Procedures

The next sections describe at high level what are the procedures involved in the identification of flaws in the drive code following the proposed methodology.

Preparation

At the preparation stage, the DD binary file structure is analyzed and loaded in the execution platform to become ready to be used at subsequent stages.

The preparation stage comprises the following steps:

- **Binary file interpretation.** Consists in the identification of the binary file format by reading the file contents and matching it with one of the supported structures, e.g., COFF, ELF or EFI. Using the appropriate file format, the process continues with decoding and locating in the binary file of all internal structures and sections, such as the machine code, data regions, relocation tables and external dependencies.
- **Binary file loading.** The binary file is analyzed and mapped in the memory of the execution platform. Each byte is linked to a metadata structure that holds information about the byte contents, such as the section where it belongs. The bytes that belong to code sections are interpreted to form instructions. Each instruction is linked to a metadata structure that represents the machine code in a higher level language (e.g., assembler), keeps track of the address location of the bytes in the executable memory, the section the instruction belongs to, and the access privilege (read, write, execute). A counter is also maintained to keep track of how many times the instruction has been executed.

The mentioned metadata guarantees that enough information exists to detect attempts to access the executable memory in the middle of an instruction during the dynamic execution. This is necessary because in CISC architectures the instructions have different sizes and it is allowed to start execution from any address (which may be the middle of a particular instruction). Therefore, it is possible to have a new set of interpreted instructions starting from the middle of a multiple byte instruction which can be useful for exploits. On the contrary, in RISC architectures this is not possible because each instruction is not spawn in multiple bytes.

- **Relocation and linkage.** Ensures that all data and code can be correctly accessed. Links to imported functions are taken care, guaranteeing that they reference the imported functions provided by the execution platform.

Binary Code Pre-processing

The second stage of the methodology uses the metadata obtained at the previous stage to perform a pre-processing of the binary code of the DD. This pre-processing builds meta data that represents the internal functions of the DD, such that they can be dynamically exercised at a later stage. The identification of the precise location of the internal functions may need to resort to several interactions because of dependencies on the target architecture, instruction set, compiler options, code optimization and the existence/absence of parameters and local variables. The reason for this is related with the prolog/epilog of each function that can differ, influenced by the previous factors, potentially leading to difficulties in locating the beginning/ending of functions in a single iteration.

As a last resort, the analysis of the DD code described at the next stage may commence without knowing the location of any internal function, except the entry point function of the DD. Starting from the entry point, and every time a call to an internal function is detected, a new round of binary code pre-processing is performed (if necessarily recursively) to identify the remaining functions.

Another objective of this stage is to identify the use of potentially insecure functions. It may not be possible at this stage to determine which internal functions make use of these potential threats because of the impossibility to correctly determine the internal function location (due to the reasons explained previously) or due to call indirections.

The final objective of this stage is to identify for each internal function all possible code paths and the decision points that form them, as soon as the internal function location is determined. This is achieved by building the execution tree that

corresponds to the function code and then follow all code paths from the root of the tree until all the branch leaf's. The tree is formed based on the following ideas (not exhaustive):

- Each instruction (not a jump and not a return instruction) is stored at the left branch of the tree;
- Whenever a conditional jump instruction is identified a node is built;
- The left branch of a node is taken if the conditional jump condition is false;
- The right branch of the node is taken if the conditional jump condition is true;
- The branch ends (a leaf is detected):
    - whenever a return instruction is identified, or;
    - whenever an unconditional jump instruction refers to an address of an instruction already existing in any of the branches from the current location up to the tree root.

Next, starting from the root of the function tree until all branch leaf's, determines all possible code path combinations. During the formation of all the code paths potential loops are also detected and noted in the metadata structures.

Supervised Emulation Analysis

The objective of the Supervised Emulation Analysis procedure is to exercise the DD binary code and determine if there are errors, where they are located and what are the conditions for triggering them. The reason for the need to execute the DD code is related with the difficulties in establishing a direct relation between the internal functions of the driver, the input parameters, return values from function calls performed inside the driver code and the driver state that can lead to flaws being triggered. These combinations may result in complex formulas that cannot be easily resolved with static analysis.

A complete driver dynamic analysis involves the verification of the compliance of the driver code with multiple OS mechanisms starting at its initialization, going through all its available services and finishing with its removal. The supervised emulation analysis starts with the invocation of the entry point function of the DD and continues with the execution through each of the exposed services. For instance, if the driver has registered dispatch functions during its initialization, then the driver IRP handling mechanism is one of the target of the analysis. On the other hand, if the driver implements the plug and play mechanism then a strategy for dynamically executing this facility should be implemented. Using this knowledge,

one can direct the analysis of the DD to target the interfaces that process external data either from unprivileged applications or from communication devices (for instance, interrupt routines) and look for errors that may be exploited.

During the emulation analysis, the Test Manager uses scripts that detail the sequence of the driver interface functions that should be tested. The Test Manager may use the facilities offered by the Execution Platform to parallelize execution of different code paths and achieve faster results. Additionally, by using automatic state snapshots of the emulated machine, which can be restored at a later time (for instance, before any conditional jump or before any call), the analysis can continue in other code paths after uncovering an error.

Whenever a Validator signals a flaw, all the information about the location of the flaw in the driver code can be reported. Irrespectively of the kind of flaw, the platform should be able to provide the faulty instruction, faulty parameters, initial conditions and sequence of events that triggered the fault.

By resorting to the information about the execution code tree and the determination of all possible code paths (determined at the previous stage), it is possible know the code coverture of the tests as well as determine potential dormant code.

Reporting and knowledge storage

The final procedure consists in reporting the encountered flaws, which includes providing information about: the involved Validators, the preconditions that triggered them, the location in the code, the involved functions, parameters and return values. A signature of a digest of the DD can also be associated with the report to form a knowledge base for future reference.

7.7 Discovery Framework

Discovery is an implementation of the Supervised Emulation Analysis methodology. The use of an emulated platform allows independency over the hardware setup usually required to test a DD. Emulation also avoids stability issues related with hangs and crashes in case of DD malfunctions in the testing platform. Through the control of the emulation machine, Discovery offers the possibility to detect errors and vulnerabilities at machine instruction level, function level and post execution level.

Discovery has granularity control over the machine code execution of the DD supporting very detailed checks at the level of each instruction execution. This way it is possible to catch platform dependent flaws such as buffer overflows, incorrect

pointer dereferences, invalid jumps and calls. In Discovery, all the functions imported by the DD are also emulated by the platform. Checks embedded at each imported function allows the detection of flaws at a higher execution level such as, incorrect handlers, incorrect pointers and invalid use of OS objects. Finally, by performing post execution checks, Discovery can find resource leakages, deadlocks and other complex conditions.

In the next sections, we are going to detail the framework with focus in the architecture components.

Architecture

Figure 7-1 depicts the architecture of Discovery. Starting from the top of the figure, an Application dynamically links to the framework and has access to the functions exposed at the Application Interface Layer. The Application provides to the users (and/or systems) an interface through which they control the behaviour of the framework. Once the DD of interest is identified, the Application passes it to Discovery for analysis using the Application Interface Layer. The DD is loaded in the framework (marked by the dashed arrow in the figure) and the analysis can start.

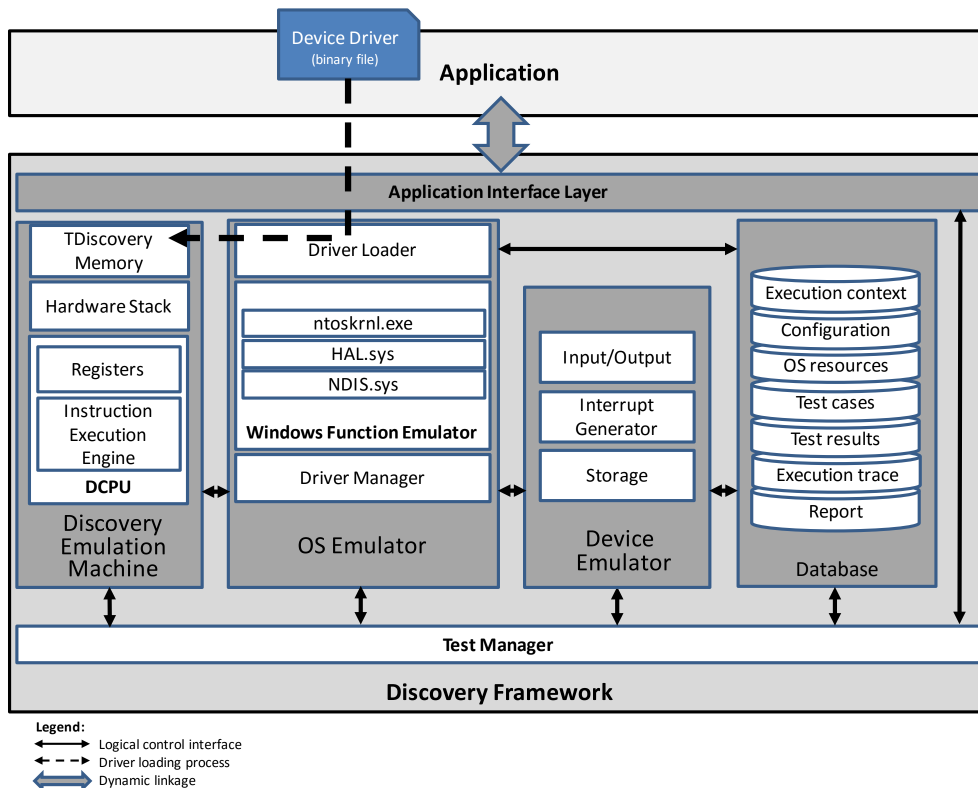


Figure 7-1: Discovery Framework Architecture.

The Discovery Emulation Machine group of components offers an environment where the DD machine instructions are analysed. The OS Emulator group provides all interfaces, mechanisms, and objects that are required by the DD from the OS. The Device Emulator enhances the abstraction of the framework by managing the input/output information, managing the interrupt generation and storage of information related to the device driven by the DD. The Database group is used to store the execution context of the framework, configuration information, the OS resources managed by the OS Emulator, test cases and results, the execution trace of the framework, and the data for reporting. All the activity of the analysis is controlled by the Test Manager which oversees the orchestration of the components.

Discovery Emulation Machine

The Discovery Emulation Machine (DEM) implements a simplified x86-64 platform where the DD code analysis occurs. It follows a modified Harvard architecture, which contains a processing unit with an arithmetic logic unit and processor registers, a control unit with an instruction register and program counter, a memory to store both data and instructions and input and output mechanisms. By implementing an x86-64 type of architecture, Discovery addresses one of the most popular computer architectures which is used in modern personal computers and servers. In any case, the approach taken by Discovery can be extended to support other types of architectures.

We have considered using existing virtual platforms such as Bochs [167] or QEmu [168], but in the end, we opted to develop our own emulation platform because of the complexity of stripping out all the unnecessary components (e.g., BIOS, IO Bus, bridges) to execute the DD code. Instead of investing time in understanding how these architectures would fit our needs and the required changes to such systems, we built something more suitable for our needs.

TDiscoveryMemory

In an x86-64 conventional system, the memory can be considered as an array of consecutive cells distinguished from each other by their address location. During the code execution, the CPU reads the memory contents pointed by the instruction register, decodes the instruction and executes the associated algorithm. In CICS CPU architectures, as it happens in x86-64, instructions may occupy more than one memory cell which may require the CPU to perform multiple memory accesses to complete the execution of one instruction.

Physical memory constraints have been resolved by resorting to mechanisms that virtualize the memory space, giving the illusion that memory is many times greater than what exists.

In our implementation, we have defined the `TDiscoveryMemory` structure to represent the executable memory of the platform (see List 7-1).

Each `TDiscoveryMemory` cell contains the address where the first byte of the machine instruction would be positioned in conventional memory, the assembly instruction already decrypted in text format, the number of parameters of that instruction and the characterization of each parameter in the instruction.

```
1    typedef struct {
2     um64 address;             //instruction address
3     char asmInstruction[50];  //decoded instruction
4     char byteCodes[20];       //raw instruction
5     int nbrParams;            //number of parameters
6     TTValue param[MAX_PARAM]; //parameters
7     int execCounter;          //number of executions
8     …
9    } TDiscoveryMemory;
```

List 7-1: TDiscoveryMemory definition (sample).

During the loading process of a DD's binary file into the DEM, the binary code is pre-processed and transformed into assembler instructions using NASM [156]. Then, a representation of that information is stored in `TDiscoveryMemory` cells.

This organization was followed for the following main reasons: it reduces the efforts on interpretation of the CPU instruction set during code analysis and code emulation execution; it maintains a metadata structure about each instruction, parameters and number of executions; it can detect attempts of executing different instructions sequences as a result of landing in the middle of variable sized instructions (a technique used to exploit the architecture of CISC architecture).

We are aware that from the point of view of memory space efficiency, `TDiscoveryMemory` is by far less efficient than the conventional x86-64 memory organization, but our objectives are quite different from just running the executable program. Additionally, since the average dimension of a DD is usually small, the use of such memory organization is consequently not a concern.

Discovery CPU

The Discovery CPU (DCPU) is an emulation of an x64 CPU architecture organized in two main components. The first component is a "C" structure where each field holds the status of the individual DCPU registers (see List 7-2). The second

component is the Instruction Execution Engine (IEE) that implements the DCPU internal mechanics and the machine instructions according to the algorithms of the various instructions. The instructions follow the descriptions found in [170] and although the current instruction set is not complete (for instance MMX instructions were not implemented) they have been proved to be enough to execute the off-the-shelf drivers from our experiments. In each step, the IEE uses the value of the instruction pointer `rip` register to locate the next instruction stored in a `TDiscoveryMemory` cell and execute the machine instruction algorithm.

```
1    typedef struct {
2     um64 rax, rbx, rcx,rdx,r8,r9,…//registers
3     um64 cpuflags;     //flags
4     um64 rbp,rsp;      //stack pointers
5     um64 rdi,rsi,rip;  //index registers
6     int cpuMode;       //operation mode
7     …
8    } DCPU;
```

List 7-2: DCPU structure (sample).

Hardware Stack

In most computer architectures, a Hardware Stack is an area of the computer memory with a fixed origin and variable size that is involved in the execution of functions, transport of parameters and allocation of local variables of functions. In Discovery, the Hardware Stack is implemented detached from the executable memory hold by `TDiscoveryMemory` cells. The Hardware Stack is simply an array of bytes managed through the `rsp` and `rbp` registers of the DCPU. Similar to what happens with a conventional x86-64 architecture, the Hardware Stack gives support to `push`, `pop`, `call` and `ret` instructions.

Operating System Emulator

The Operating System Emulator (OSE) is the functional interface to the DD. The OSE is: i) in charge of loading the DD and maintain data structures that support the DD execution, ii) provide all the imported functions called by the DD, and iii) call the DD call-back functions using the appropriate function signatures and parameters. These three main tasks are provided respectively by the Driver Loader, the Windows Function Emulator and the Driver Manager components. The following sub sections explains the implementation of these components and the role they perform in the overall architecture.

Driver Loader

In the Windows OS, a DD installation package usually contains at least an "`.inf`" and one or more "`.sys`" files. The "`.inf`" file is a text based file organized into several sections used during the installation of the DD in the system. The OS employs this file to match devices with drivers whenever a new device is found in the hardware platform. The "`.inf`" file contains information about the appropriate "`.sys`" filename to be used to drive the device.

The "`.sys`" file is the binary image of the DD and contains the machine instructions that must be loaded in memory to execute and control the device. It follows the PEF [32] format for the file structure, the same utilized by applications and DLLs, which includes in a single file the machine code of the DD and dependences from other software modules organized in the form of tables. The imported functions table contains the name of the functions and the name of the external modules (DLLs or other software modules) from which the DD depends. The OS uses this information during the software loading process to link the DD code to other software modules necessary for correct execution. In some cases, the required modules may not yet be present in the system. When this happens, the OS has to perform additional loadings that may result in some kind of recursion process.

In Discovery, the Driver Loader (DL) is the component responsible for the loading process of the DD intended for analysis in the emulation machine. The loading process is performed in two phases:

- **Phase 1 – File read and preparation:** The DL reserves temporary regular memory space in the Discovery application and reads the "`.sys`" file to that memory. Following the specification of the PEF format, the DL interprets the contents of the temporary memory and locates the various sections. The code section is prepared for execution by fixing the relocation addresses and linking the imported functions discriminated in the import section table to the functions provided by the Windows Function Emulator. At the end of phase 1, the DL has an image of the DD loaded in temporary memory where all imported functions used by the DD are already linked to the functions provided by the framework;
- **Phase 2 – Building the executable memory contents:** The DL walks through the temporary memory to disassemble the machine instructions in the code section. For each instruction, the DL allocates and builds a `TDiscoveryMemory` memory cell with the corresponding metadata. During this process, the existing internal functions are identified by matching the processed instructions with the prologue and epilogue machine instruction

sequences that form the start and end of functions. To complement this identification, call instructions are interpreted and the destination address identified. Destination addresses embedded in the instruction (e.g., `call dword [dword 0x0800ABCD]`) are easy to check for either an internal function or an imported function. New previously unidentified internal functions are then dynamically formed. Indirect calls (e.g., `call esi`) should be checked later.

Figure 7-2 illustrates the memory organization of a `TDiscoveryMemory *discoveryMemory` array used to represent the binary code of a DD. As an example, the cell `discoveryMemory[0]` represents the first instruction of the binary code and `discoveryMemory[1]` represents the second instruction of the binary code. Each of these cells already contains a series of metadata necessary for the DEM to execute.

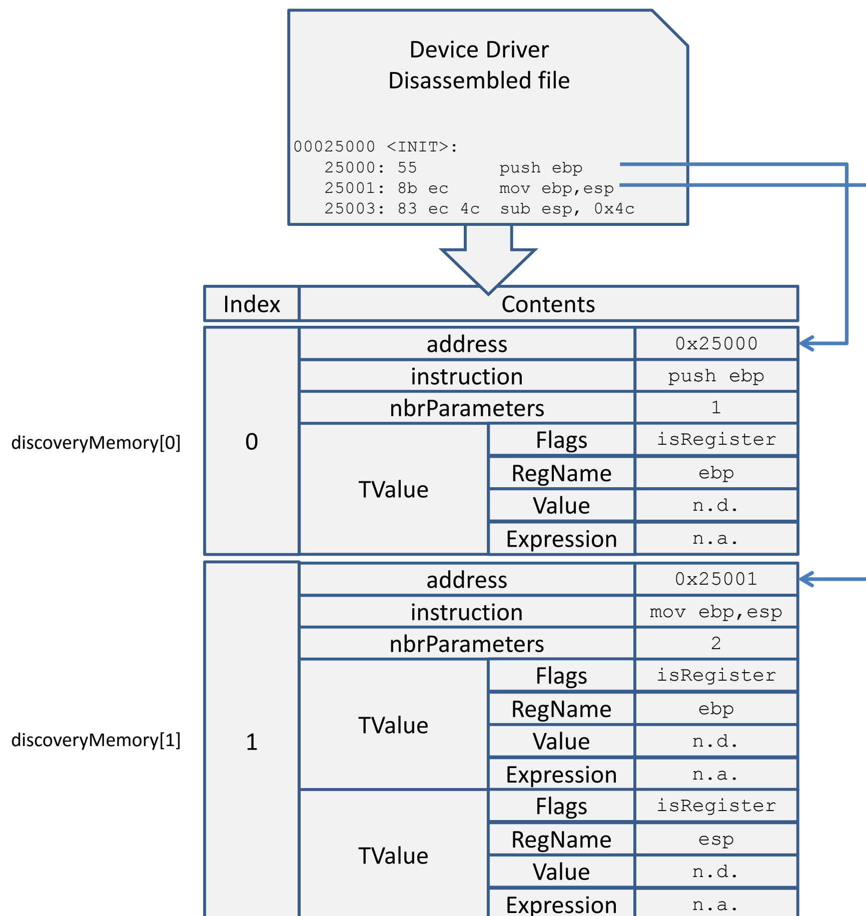


Figure 7-2: Example of Discovery Memory organization.

Windows Function Emulator

A Windows DD depends on functions provided by the OS. These functions are described in the DDK, and form the API provided by the OS to the DD. They are used by the DD to register the call-back functions in the OS, to request and free resources from the OS and to perform various other operations.

Table 7-1: Summary of Implemented Windows Emulator functions.

Import library	Function group	Description	File modules	Number of Functions
Ntoskrnl.exe	I/O manager	Service functions	NtosKrnlIoManagerRtl	16
	Memory manager	Page table control	NtosKrnlMemortyMgr	6
	Executive	Heap management and synchronisation	NtosKrnlExecutiveLib	11
	Power Management	Power control	NtosKrnlPoMgr	4
	Runtime (prefix Rtl)	Utility and management routines replacing ANSI-standard routines	NtosKrnlRtl NtosMutextInterface NtosKrnlMgr NtosKrnlList	22
	Zw routines	File and registry access	ZwXxxRoutines	10
	Windows kernel	Low-level synchronization functions	NtosKrnlCoreKrnlLibRtl	32
Hal.sys	Hardware abstraction layer	Provides an abstraction of the hardware	Hal	23
Ndis.sys	Network driver interface	Network support	Ndis	148
WMiLib.sys	IoManager	Windows Management Instrumentation	WMI	2

In Discovery, the Windows Function Emulator (WFE) is the module that implements the functions listed in the "`.import`" section of the DD. Table 7-1 gives a summary of the type and number of the currently implemented functions that can be linked to the DD.

The WFE defines the `TFuncTranslation` structure (see List 7-3) to establish the correspondence between the name of an imported function (`fxName`) and the address of the corresponding function implemented at the WFE (`*_My_fxAddr`). Other attributes such as the calling convention (`callingConvention`) and the number of parameters (`nbrParams`) of the function are also represented in the structure. All Windows functions implemented in WFE are arranged in an array of

TFuncTranslation elements which is used during the linkage process of the DUT to correctly locate the address of imported function and connect the imported functions of the DD with Discovery.

```
1  typedef struct{
2   char fxName[255];     //name of the function
3   DWORD *_My_fxAddr;    //address of the function
4   int callingConvention;//function calling convention
5   int nbrParams;        //Number of parameters
6  }TFuncTranslation;
```

List 7-3: TFuncTranslation – Linkage of imported functions.

Driver Manager

The Driver Manager (DM) is the component of the OS Emulator in charge of invoking the DD interface functions and maintaining the resources that the DD requires from the OS Emulator for execution. It is for instance the Driver Manager that holds the `struct _DRIVER_OBJECT *DriverObject` parameter on the call to the `DriverEntry` function. Besides maintaining all the necessary structures, it is the DM that passes the parameters to the DD according to the calling convention in use by the target platform (either using the Hardware Stack or the registers) and setups the registers of the DCPU such that the execution context can switch to the DEM and the code of the DD.

Device Emulator

It becomes challenging to emulate any device without knowing the details of the hardware. For instance, the DD may look for specific values read from a specific port to determine its state and continue operation. The hardware independency is achieved by ensuring that the code paths depending on `in` and `out` instructions are covered, something delegated to the Test Manager. More sophisticated devices use abstract ways to deal with input/output. In these cases, the Device Emulator interprets and processes complex structures such look-a-side buffers and DMA memory representations, which typically occurs with most of the modern DD that deal with PCI, USB and NDIS specifications. In this case the import functions provided by the OS Emulator are involved

The Device Emulator manages information related to input/output requests performed by the DD code whenever it interfaces with the hardware either directly using `In` and `out` instructions or indirectly when intermediated by the OS Emulator.

The objective of this component is not to emulate a replica of the device managed by the DD (it will be impossible since Discovery does not know which device is involved), but rather to provide the mechanisms to analyse the DD code.

Database

Discovery uses a database to keep track of test cases and test results. The database also maintains the content of the execution of the DEM, the resources managed by the OS Emulator and the information to generate reports. This way, Discovery ensures that it contains all the data to be able not only to reproduce results, but also to accurately report the detected flaws.

Test Manager

The Test Manager is the component in charge of supervising the strategy employed to find the errors in the DD code. It uses the internal structure of the DD under test to dynamically generate the test cases and implement a testing strategy (see section 7.9).

7.8 Discovery Emulation Execution Mechanisms

This section describes a few mechanisms that glue all the components of the framework enabling the analysis of the DD.

Execution Context Switch

The DEM has two main modes of operation distinguished by the code that is being executed. The DEM is running in *emulation mode* when a DD function is being executed. The DEM is running in *true mode* when the DD calls a WFE function, a DD function execution finishes or a flaw in the DD has been detected. Whenever a change from *true mode* to *emulation mode* occurs (and vice versa) it is said that an execution context switch has occurred. It is important to understand in which execution mode the DEM is running to comprehend what are the techniques involved in the detection of DD flaws.

Calling DD Interface Functions

DDs comply with a defined structure and, as explained before, the `DriverEntry` function is the entry point to the driver code. The DD exposes other functions either by filling in the address of the call-back functions in the `DRIVER_OBJECT` data structure (when `DriverEntry` returns) or by registering call-back functions to the

OS using appropriate registration functions, such as, `NdisMRegisterMiniportDriver` in the case of a NDIS DD.

The Driver Manager is the component of Discovery that directly calls the DD Interface functions. When a DD function is called, there is a switch on the execution mode of DEM from *true mode* to *emulation mode*. The switching algorithm can be described as follows:

- Determine which function of the DD to call and obtain the signature of the DD function;
- Prepare the parameter values and pass the parameters to the DEM according the type of execution platform (i.e., 32 bit or 64 bit);
- Force the return address in the Hardware Stack to a Driver Manager function, ensuring that when the DD function ends the DD switches the context of the DEM to *true mode* in a controlled way;
- Setup the `rip` register value of the DCPU to the address of the DD function to be executed;
- Enter into emulation mode by transferring the execution control to the DCPU with a call to `cpu_run()` function.

Although the call of the DD functions is performed by the Driver Manager, it is the Test Manager that instructs it. The DEM continues to run in *emulation mode* until one of the following events occurs:

- The DD calls a WFE function;
- The DD code execution finishes by returning the execution to the address of the Driver Manager entry function;
- A flaw is detected by one of the validators during the computation of a binary instruction.

Executing WFE Functions

The DEM executes the DD code in *emulation mode*. Whenever a `jmp`, `call` or `ret` instruction targets the address of a WFE function, the execution of the DEM changes from *emulation mode* to *true mode*. The algorithm of this context switch is implemented at the `cpu_step` function and can be described as follows:

- Obtain the next instruction address and verify if it refers to a `TDiscoveryMemory` cell:
    - In the affirmative case, continue the execution at that address;

- Otherwise, verify if the address belongs to a WFE function. In this case perform the context switch by calling `cpu_executeWFEFunction`;
  else, raise a flaw exception.

Returning Control to Driver Manager

Under normal circumstances, when the execution of a DD function ends, the Driver Manager entry point function is called. When this happens, the Driver Manager returns control to the Test Manager so that it decides what should be the conditions to perform the next test.

On the contrary, if a flaw is detected, the Driver Manager entry point will not be called. The emulation (or the execution of a WFE function) will end because one of the Validators signals a fault event to the Test Manager.

7.9 Detection of Flaws

This section describes the mechanisms involved in the detection of flaws, how they are triggered and what kind of flaws it is possible for Discovery to find. We start this section by presenting Primitive Checkers, a set of functions responsible for the detection of basic errors in the DD code. These are the building blocks for constructing more complex verifications. Then, we explain the Validators embedded in Discovery and group them in two different classes. Next, we present the adopted testing strategy. Finally, we conclude the section by enumerating the type of flaws that can be detected with the currently implemented Validators.

Primitive Checkers

In Discovery, a primitive checker is a function that evaluates an input parameter and returns `true` or `false` depending if the parameter satisfies or not the success criterion. Table 7-2 includes the list of currently implemented primitive checkers.

As an example, primitive checker PC3, `isValidStackAddr(um64 address, int range)`, returns `true` if the parameter `address` and `address+range` is within the range of the Hardware Stack. This primitive is suitable to check if a certain `address` is a plausible local variable. In this checker, the `range` parameter gives the possibility to check an interval of consecutive addresses starting at `address`. This is the basic mechanism for buffer overflows detection in the Hardware Stack, as a result of consecutive `mov` instructions (including `movsd`, `movsw`, `movsx`).

Another example, PC4, `isValidWFEAddress`, verifies is the `address` parameter corresponds to the address of a function provided by the Windows Function Emulator. This primitive checker is useful to verify if a call to the specified address parameter can be performed.

Table 7-2: List of Primitive Checkers.

ID	Name	Input Parameter	Description
PC1	isValidSourceRegister	*instruction, regName*	Returns true if *regName* is a valid instruction source register operand.
PC2	isValidDestinationRegister	*instruction, regName*	Returns true if *regName* is a valid instruction destination register.
PC3	isValidStackAddr	*address, range*	Returns true if *address* and *address+range* belongs to the address interval of the Hardware Stack.
PC4	isValidWFEAddr	*address*	Returns true if *address* corresponds to an address of a WFE function.
PC5	isValidDriverManagerAddr	*address*	Returns true if *address* is the address of the entry point of the Driver Manager.
PC6	isValidTDiscoveryCell	*address*	Returns true if *address* is an address of a TDiscoveryMemory cell.
PC7	isValidMemoryFromOS	*address, range*	Returns true if *address* and *address+range* belongs to memory managed by the OSE.
PC8	isValidOSObjectHandler	*Handler*	Returns true if *handler* is and identifier provided by the OSE.
PC9	isValidDataSegment	*address, range*	Returns true if *address* and *address+range* belongs to the address interval of the DD data segment.

Validators

Discovery uses Validators during the DD code analysis to perform a check over an intended action. The output value of a Validator may be `true`, which means that no flaw was detected and the intended action is harmless, or `false`, which indicates that a flaw has been found.

Table 7-3 presents the list of the currently implemented Validators and flaws that can be detected. The first column, contains the identifier of the Machine Level Validator (MLV), Function Level Validator (FLV) and Post Execution Validator (PEV).

Column “Name” gives a designation to the Validator and establishes an implicit relationship between the name and the target of the check that is performed. Column “Flaw” describes the type of flaws that the Validator can detect. Column “Possible Causes” gives a non-exhaustive list of possible causes for the flaw.

Finally, the last column gives a non-exhaustive list of the possible consequences of not catching the flaw.

Validators are built using Primitive checkers. For instance, MLV1 Source operand is built using PC1, PC3, PC4, PC5, PC6, PC7, PC8 and PC9.

Table 7-3: List of implemented validators and detectable flaws.

ID	Name	Flaw	Possible causes	Possible consequences
MLV1	Source operand	Invalid source operand	• Uninitialized variable • Corrupted pointer	• Buffer overflow • Hang • Crash
MLV2	Destination operand	Invalid destination operand	• Uninitialized variable • Corrupted pointer	• Buffer overflow • Hang • Crash
MLV3	Call, jmp and ret destination address	Invalid address for execution	• Uninitialized variable • Corrupted pointer	• Privilege elevation • Hang • Crash
MLV4	Unconditional jump destination address	Invalid address for execution	• Corrupted pointer	• Privilege elevation • Hang • Crash
FLV1	MemoryRange	Invalid address	• Uninitialized variable • Corrupted pointer	• Buffer overflow • Hang • Crash
FLV2	Handler	Invalid handler	• Uninitialized variable • Corrupted pointer	• Hang • Crash
FLV3	ParameterRange	Invalid value for parameter	• Uninitialized variable • Corrupted pointer	• Hang • Crash
FLV4	DeadLock	Dead lock	• Uninitialized variable • Corrupted pointer • Incorrect control of resources	• Hang • Crash
FLV5	IRQL	Invalid IRQL for function	• Invalid function context control	• Hang • Crash
FLV6	Return Value Evaluation	Non validation of function return value	• Incorrect function context control	• Hang • Crash
FLV7	Explicit call to crash function	Bug check function called	• Explicit call from the DD to a function that crashes the OS.	• Crash
PEV1	ResourceLeakage	Resource leakage	• Uninitialized variable • Corrupted pointer • Incorrect control of resources	• Hang • Crash
PEV2	DormantCode	Code dormant	• Compilation errors • Backdoors	• Hang • Crash • Disclosure of confidential information

Testing Strategy

The Test Manager uses the internal structure of the DD under test to dynamically generate the test cases based on: 1) The entry point of the driver; 2) The remaining interface functions exposed by the DD to the OS (registered by the DD during its execution); 3) The possible return values/output parameters of the imported

functions called by the DD; and 4) The documented calling sequences performed by the OS to the interface functions of the DD.

The objective of the Test Manager is to execute all test cases defined for an interface function, import function (i.e., output/return values) and be able to either reach the end of the execution or to find an error.

The test campaign starts at the `DriverEntry` function and once the tests determined for this function are finished, the Test Manager moves to another interface function exposed by the DD to the OS. For this, the Test Manager uses the functions that were registered by the DD using the `MajorFunction` array of the `Driver_Object` parameter of `DriverEntry` or by calling specific OS functions (either still in the execution context of the `DriverEntry` function or in the execution context of other interface functions).

The order used by the Test Manager to test each interface function mimics the way the OS uses such functions, thus avoiding sequences that do not make sense for the DD. Otherwise, if executed, these sequences could lead to false positives (for instance, calling the `AddDevice` function after calling the `DriverUnload` function).

Since the interface functions exported by the DD to the OS are known and documented, it is possible to build calls to these functions with diverse parameter values that should be handled correctly by the DD.

While testing an interface function, whenever a call is performed to an internal function of the DD, the Test Manager changes the test focus and initiates the test campaign of such function. This happens recursively, until the Test Manager finds an internal function that does not call any internal function. Whenever it finishes the test campaign of an internal function it changes the focus to the preempted testing function.

The Test Manager maintains control over each call performed by the DD code to external functions keeping track about each code path where the call was performed, what was the returned result and the value of the output parameters. This way, the Test Manager can run diverse tests within an internal function of the DD and change the return value (or output parameter value) of any called WFE function in each test case.

Tests Cases

From the point of view of the tests, Discovery interfaces the DUT at two different levels: i) at the DUT interface functions and ii) at the provision of the OS functions (in this case the functions implemented by the WFE component). Therefore, the

following group of test cases can be identified: i) driver interface test cases (DITC) acting at the DUT' interface and ii) imported function test conditions (IFTC) that control the return values and output parameters of the WFE functions. The selection and usage of each of the test cases is controlled by the Test Manager during the analysis of the DUT.

Driver Interface Tests Cases

Table 7-4 represents the test cases at the driver interface of the DUT. DITC1 represents a normal situation where the Test Manager calls a function of the DUT with valid parameters. DITC2 represents a situation where the Test Manager passes invalid parameters to the DUT. Although DITC2 represents an uncommon situation (because typically the OS does not pass invalid parameters to the DD) it was included to demonstrate the level of dependency that usually DD have from the OS in what regards to the correctness of the input parameters.

Table 7-4: DITC test values.

ID	Parameter passed to DUT function	Description
DITC1	Valid value	The Test Manager passes valid parameters to an interface function of the driver.
DITC2	Invalid value	The Test Manager passes invalid parameters to an interface function of the driver.

Import Function Tests Conditions

An imported function falls into one of the following signatures: i) have no return value and no output parameters, ii) have return value and no output parameters, iii) have no return values but have output parameters and iv) have both return value and output parameters. Table 7-5 represents the applicable test cases that simulate the possible outcomes on the usage of the imported functions called by the DUT during its operation. For instance, IFTC1 represents a situation where function `Fx` called by the DD has a successful outcome. On the contrary, IFTC2, represents a situation where function `Fx` had an unsuccessful outcome. Naturally, functions that do not return values and do not have output parameters are not considered for test conditions. In these cases, the Test Manager has to guarantee the correct outcome for the tests to be meaningful.

Currently, Discovery has over 260 imported functions defined and over 390 imported functions test conditions in its database.

Table 7-5: IFTC combination values.

ID	WFE return value	WDE output parameter	Description
IFTC1	Success	No output parameters	The DUT calls a WDE function and the WDE function return value informs successful execution. The WDE function does not have any output parameters.
IFTC2	Fail	No output parameters	The DUT calls a WDE function and the WDE function return value informs an unsuccessful execution. The WDE function does not have any output parameters.
IFTC3	Success	Min Valid Value	The DUT calls a WDE function and the WDE function return value informs successful execution. Output parameters have minimum value for the involved type.
IFTC4	Success	Valid Value	The DUT calls a WDE function and the WDE function return value informs successful execution. Output parameters are valid (e.g., memory allocation pointers are valid).
IFTC5	Success	Max Valid Value	The DUT calls a WDE function and the WDE function return value informs successful execution. Output parameters have maximum value for the involved type.
IFTC6	Success	Invalid Value	The DUT calls a WDE function and the WDE function return value informs successful execution. Output parameters have invalid values.
IFTC7	Fail	Invalid value	The DUT calls a WDE function and the WDE function return value informs an unsuccessful execution. Output parameters contain values susceptible to cause problems if used (e.g., NULL pointers).
IFTC8	No return value	Valid value	The DUT calls a WDE function and the WDE function does not have a return value. The output parameters contain values usable by the DD (i.e., not susceptible to cause any problem, e.g. memory allocations are valid).
IFTC9	No return value	Invalid value	The DUT calls a WDE function and the WDE function does not have a return value. Output parameters informs unsuccessful execution of the function (e.g., NULL pointer in memory allocations).

Test Sets

In a real environment, the way that the OS calls the exposed interface of the DD is not arbitrary. Although it depends on external events, it follows a specific pattern. Therefore, for the analysis of the DD to be meaningful, whenever dynamically emulating the execution of the DUT code, the Test Manager must mimic the OS sequence of calls. Otherwise, in most the cases calling the DD interface arbitrarily can lead to false positive results.

Table 7-6 presents the considered test set used by the Test Manager for the experiments. The applicability of each of the sequences is determined by the Test Manager and dependent on the exposed interface of the DUT.

Table 7-6: Applicable call sequence test conditions (not exhaustive).

Sequence ID	Call Sequence
S1	• DriverEntry • DriverUnload
S2	• DriverEntry • AddDevice • DriverUnload
S3	• DriverEntry • AddDevice • IRP_MJ_XXXX functions • DriverUnload
S4	• DriverEntry • AddDevice • Ndis initialization routines • IRP_MJ_XXXX functions • Interrupt Routines • DriverUnload

S1 represents the case where the DD is installed in the OS and is removed immediately. In S2, the `AddDevice` function is called after `DriverEntry`, and the DD removed right after. In S3, the `IRP_MJ_XXXX` functions are called after `AddDevice`. Calling `IRP_MJ_XXXX` functions without `AddDevice` can lead to errors because the `IRP_MJ_XXXX` function may try to access the `driverExtension` fields before it has been created (typically) in `AddDevice` function. S4 contains calls to interrupt routine functions. The sequences can have more complex combinations, but the current version of Discovery only contains these ones for now.

At the end of each of the test sets, the Test Manager can assess the balance of the resources used by the DUT and determine situations that can be caught by Post Execution Validators.

Expected failure modes

The detection of errors during the analysis is performed using the Validators described previously. In Discovery, there is a one to one correspondence between the Validators and the expected failure modes as structured in Table 7-7.

Table 7-7: Expected failure modes.

ID	Flaw	Validator
FM-MLV1	Invalid source operand in instruction.	MLV1
FM-MLV2	Invalid destination operand in instruction.	MLV2
FM-MLV3	Invalid address for execution in call, unconditional jump and return instructions.	MLV3
FM-MLV4	Invalid address for execution in conditional jump.	MLV4
FM-FLV1	Invalid address passed to WFE function.	FLV1
FM-FLV2	Invalid handler passed to WFE function.	FLV2
FM-FLV3	Invalid value for parameter.	FLV3
FM-FLV4	Dead lock.	FLV4
FM-FLV5	Invalid IRQL for function.	FLV5
FM-FLV6	Non validation of function return value.	FLV6
FM-FLV7	Explicit call to crashing function	FLV7
FM-PEV1	Resource leakage.	PEV1
FM-PEV2	Dormant code.	PEV2

Implementation

Discovery is a framework whose components can be reused to build other tools for the detection of flaws in DD. The framework was implemented using Visual Studio 2013 and is written in assembly, “C”, “C++” and “C#” languages. It is made available in the form of a Dynamic Linking Library (DLL). Using the “`Discovery.dll`” it is possible to create a graphical user application (and web services) that receives as input a DD binary file, performs the analysis and returns a report about the potential presence of flaws.

Table 7-8, gives an estimation of the lines of code of the latest version of Discovery [169]. The C++, C and assembly files form the core of the platform. Many of the C/C++ Header files were built based on Microsoft’s DDK code. The C# code belongs to the user interface and the make files are automatically managed by the Visual Studio.

Table 7-8: Count of Lines of Code of Discovery.

Language	Files	Lines of Code
C++	231	141,631
C	225	125,542
C/C++ Header	515	99,657
Assembly	154	21,284
C#	52	7,647
Make	17	2,142

7.10Experimental Results

In this section, we present the test conditions and the results of the experiments performed with drivers included in the installation disks of commercially available products. The experiments were performed in a laptop computer HP Pavilion with an AMD A8-6410 APU, 2.00GHz, with 6.00GB memory and an 220GB SSD Toshiba Disk. Each of the driver under test (DUT) were subject to a series of situations that simulate possible execution conditions in a real environment. If errors exist during the execution, they are caught by the action of the Validators of Discovery.

These drivers were selected taking into consideration the current development stage of Discovery and their relative simplicity, although, commercially available.

Experiments with a Bluetooth Driver

The first set of experiments targeted the `btwrchid.sys` (BT) HID Bluetooth controller driver found as part of the installation package of the ASUS USB-BT400 Advanced Bluetooth 4.0 Adapter for Windows 10.

Table 7-9 summarizes the characteristics of the BT DD. The size in disk of BT is 20,480 bytes and the code is organized in 6 different sections. The `.text` section, which contains the machine instructions, is 7,552B length which translated to 3,265 different instructions stored in TDiscoveryMemory cells.

During the loading process, Discovery found that BT1 imported 37 functions from the OS, 32 from the `ntoskrnl.exe`, 3 from `hal.dll` and 2 from `hidclass.sys`.

Discovery detected 42 internal functions which evolved to 45 at the end of the analysis process. This difference confirms that a simplistic analysis on the driver code based on the detection of prolog and epilog of functions is usually not enough to be able to detect the overall existing functions. To avoid this inaccuracy, during the execution of the DD code, Discovery dynamically detects and considers for analysis previously undetected internal functions.

Table 7-9: Characteristics of the BT DD.

Characteristic	Value
ID	BT
Device driver file name	btwrchid.sys
Type	Bluetooth HID Controller
Vendor	Broadcom
Target OS	Windows 10
Target Platform	32 bit
File size in disk	20,480B
.text section	Start: `0xB8A0480`, Size: 7,552B
.rdata section	Start: `0xB8A2200`, Size: 384B
.data section	Start: `0xB8A2380`, Size: 256B
INIT section	Start: `0xB8A2480`, Size: 1,152B
.rsrc section	Start: `0xB8A2900`, Size: 1,024B
.reloc section	Start: `0xB8A2D00`, Size: 512B
DriverEntry address	`0xB8A24BE`
Number of TDiscovery memory cells	3,265
Number of imported functions from ntoskrnl.exe	32
Number of imported functions from hal.dll	3
Number of imported functions from hidclass.sys	2
Initial number of local DD functions	42
Final number of local DD functions	45

Imported Functions Test Cases for BT

After loading the BT DD, and based on the imported functions used by the DUT, the Test Manager automatically selects the applicable test cases to be used whenever the DUT calls any of the imported functions. Table 7-10 shows the test cases for each of the eligible WFE functions used by BT DD. As an example, for the `PVOID ExAllocatePoolWithTag (_In_ POOL_TYPE PoolType, _In_ SIZE_T NumberOfBytes, _In_ ULONG Tag)` function, two possible conditions are considered: i) IFTC1 the function succeeds and returns a valid pointer and ii) IFTC2 the function fails and returns `NULL`. Since the function does not have any output parameters, IFTC3 to IFTC9 are not applicable to this function. To avoid false positives, IFTC6 is not considered in our tests.

Although BT uses many other imported functions that carry return values and/or output parameters, they were not eligible to be used for test purposes. For these imported functions the DUT cannot determine the correctness of the return/output and, therefore, an incorrect return value/parameter would potentially lead to false positives.

Table 7-10: Imported functions test cases for BT.

BT Imported Functions	Test Cases								
	IFTC1	IFTC2	IFTC3	IFTC4	IFTC5	IFTC6	IFTC7	IFTC8	IFTC9
ExAllocatePoolWithTag	✓	✓				-			
HidNotifyPresence	✓	✓				-			
HidRegisterMiniportDriver	✓	✓				-			
IoAcquireRemoveLockEx	✓	✓				-			
IoAllocateIrp	✓	✓				-			
IofCallDriver				✓		-	✓		
KeCancelTimer				✓		-	✓		
KeDelayExecutionThread	✓	✓				-			
KeSetEvent				✓		-	✓		
KeSetTimer				✓		-	✓		
KeWaiForSingleObject	✓	✓				-			
MmMapLockedPagesSpecifyCache				✓		-	✓		
PoCallDriver				✓		-	✓		
RtlInitUnicodeString						-		✓	✓
ZwClose	✓	✓				-			
ZwOpenKey				✓		-	✓		
ZwQueryValueKey			✓	✓	✓	-	✓		

✓ Tested condition

As an example, Table 7-11, lists three of these imported functions. For instance, function `LONG __cdecl InterlockedExchange(_Inout_ LONG volatile *Target, _In_ LONG Value)`, contains a target input/output parameter and returns the value of the target variable.

At the first glance, it looks like a candidate function for using IFTC4 and IFTC6, however, the purpose of this function is to atomically exchange the values of the parameters. Subverting this purpose, which cannot be verified by the DUT, would constitute an error from the OS, that potentially would lead to a false positive when applying the IFTC6 test condition.

The function `KIRQL KeGetCurrentIrql(void)` returns to the DD the current IRQL value (managed by the OS). Changing this result (arbitrarily) would lead to false positive errors.

Finally, an error in `VOID KeInitializeTimer(_Out_ PKTIMER Timer);` would affect the PKTIMER opaque structure with little or no consequences to our tests.

Table 7-11: Discarded import functions test cases for BT (not exhaustive).

Imported Functions	Discard reason
`LONG __cdecl InterlockedExchange(_Inout_ LONG volatile *Target, _In_ LONG Value);`	Sets a variable to the specified value as an atomic operation. The function returns the initial value of the target variable.
`KIRQL KeGetCurrentIrql(void);`	The KeGetCurrentIrql routine returns the current IRQL which is maintained by the OS.
`VOID KeInitializeTimer( _Out_ PKTIMER Timer );`	The KeInitializeTimer routine initializes a timer object. Timer is a pointer to a timer object, for which the caller provides the storage.

Test Cases for BT at Driver Interface

At the beginning of the test execution, the Test Manager only knows the address of the `DriverEntry` function. As the tests progress, and at the end of a successful execution of `DriverEntry`, other interface functions are registered by BT in Discovery. Table 7-12 shows the driver interface functions found during the analysis process and the generated test cases.

Table 7-12: BT Driver Interface and test cases.

Target Interface Function	Test Case ID	Test Case
DriverEntry	BTDI_TC01	Valid driverObject
	BTDI_TC02	Invalid driverObject
AddDevice	BTDI_TC03	Valid driverObject
	BTDI_TC04	Invalid driverObject
	BTDI_TC05	DeviceExtension = NULL
	BTDI_TC06	Dimension of Device Extension lower than expected
IRP_MJ_POWER	BTDI_TC07	Valid deviceObject
	BTDI_TC08	Invalid deviceObject
DriverUnload	BTDI_TC09	Valid deviceObject
	BTDI_TC10	Invalid driverObject
IRP_MJ_CLOSE	BTDI_TC11	Valid deviceObject
	BTDI_TC12	Invalid deviceObject
IRP_MJ_INTERNAL_DEV_CONTROL	BTDI_TC13	Valid deviceObject
	BTDI_TC14	Invalid deviceObject
IRP_MJ_SYSTEM_CONTROL	BTDI_TC15	Valid deviceObject
	BTDI_TC16	Invalid deviceObject

Test Results for BT

The Test Manager generates the test sets using the information about the exposed interface of the BT DD (see Table 7-12) and information about calling sequences combination present in the database. Table 7-13 gives examples of the generated test sets for this DUT. The first column identifies the test set using the BT_TSx nomenclature, where, BT is the identifier of the DUT, TS stands for Test Sequence

and x is the sequential number of the test set. The second column describes the interface functions that are tested and the sequence of their call. The third column indicates the total number of test cases generated by Discovery for the test set. The last two columns present the partial and total execution time of the experiments in seconds.

Column named “Partial” refers to the time taken to analyse the code of each of the individual interface function in the test set. The column “Total” refers to the sum of the time spent in the analysis of the execution of all the interface functions that belong to the same test set.

Table 7-13: Example Test Set for BT and execution time.

Test Set	Call Sequence	Total Test Cases	Execution Time (s)	
			Partial	Total
BT_TS1	DriverEntry	4	9,1	10,1
	DriverUnload	1	1,0	
BT_TS2	DriverEntry	1	1,8	23,7
	AddDevice	5	20,9	
	DriverUnload	1	1,0	
BT_TS3	DriverEntry	1	1,8	17,1
	AddDevice	1	4,2	
	IRP_MJ_POWER	12	10,1	
	DriverUnload	1	1,0	
BT_TS4	DriverEntry	1	1,9	720,9
	AddDevice	1	4,2	
	IRP_MJ_INTERNAL_DEV_CONTROL	101	713,8	
	DriverUnload	1	1,0	
BT_TS5	DriverEntry	1	1,9	10,7
	AddDevice	1	4,2	
	IRP_MJ_SYSTEM_CONTROL	5	3,6	
	DriverUnload	1	1,0	
BT_TS6	DriverEntry	1	1,9	9,5
	AddDevice	1	4,1	
	IRP_MJ_CLOSE	3	2,5	
	DriverUnload	1	1,0	

The number of the generated test cases results from the identified interface functions and the imported functions used during the execution of the DD. As an example, we are going to analyse the tests in BT_TS1 (the remaining test sequences, BT-TS2 to BT-TS6, follow the same principle).

The BT_TS1 test set contains a call to `DriverEntry`, followed by a call to `DriverUnload`. This represents the situation where the DUT is installed and then uninstalled in the OS. Two test cases, BTDI_TC01 and BTDI_TC02, were generated to test `DriverEntry` (see Table 7-12). During the execution of `DriverEntry`, the import function `HidRegisterMiniportDriver` is called (dynamically determined during the analysis of the DD). Looking up to Table 7-10, it shows that IFTC1 and IFTC2 were generated for `HidRegisterMiniportDriver`. Therefore, to test

DriverEntry a total for 4 test cases were generated (even though during other imported functions may be used – but no test cases have been generated for them).

Similarly, two test cases were generated for the DriverUnload interface function: BTDI_TC09 and BTDI_TC10. But since no imported functions are called by DriverUnload only these two testes have been considered for this interface function.

Considering that: i) Discovery discards test combinations that represent the same test conditions (which may happen when calling some interface function sequences, e.g., call DriverEntry and then call DriverUnload) and ii) Discovery does not apply all possible combinations of the generated tests and assumes independency over the interface functions, a total of five test cases are grouped in BT_TS1 as represented in Table 7-14.

Finally, to avoid the repetition of the same test situations over different test sets the Test Manager does not analyse interface functions that have been analysed in previous sequences (i.e., does not present potential error situations). This is the reason why DriverEntry and other functions only have one test situation in some of the test sets.

Table 7-14: Detail of BT_TS1 Test Set

Test	DriverEntry	HidRegisterMiniportDriver	DriverUnload	Note
1	BTDI_TC01	IFTC1	BTDI_TC09	
2	BTDI_TC02	IFTC1	BTDI_TC09	
3	BTDI_TC01	IFTC2	BTDI_TC09	
4	BTDI_TC02	IFTC2	BTDI_TC09	
5	BTDI_TC01	IFTC1	BTDI_TC09	Not considered since it is the same as Test 1
6	BTDI_TC01	IFTC1	BTDI_TC10	

The execution of the identified Test Sets for the BT DUT resulted in the detection of the errors summarized at Table 7-15. In the next paragraphs, we are going to detail the obtained results.

The BT_E1 error occurs when the Test Manager passed an invalid driverObject parameter to the DriverEntry function. The error was signalled by the MLV1-SourceOperand validator that was triggered when the DUT (while in InternalFunction_0005) tried to use the ecx register to access the stack and no valid memory existed at the referenced position.

The BT_E2 error occurs when the Test Manager passed an invalid driverObject parameter to the AddDevice function. The error was signalled by the MLV1-SourceOperand validator that was triggered when the DUT (while in

AddDevice function) tried to access the stack with the ebx register and no valid memory existed at the referenced position.

Table 7-15: Test Results for BT.

Error ID	Test Case ID	Failure Mode	Error Description
BT_E1	BTDI_TC02	MLV1	Invalid driverObject passed to DriverEntry. FM-MLV1 acted at: mov dword [ecx+0x74], 0xa6506de
BT_E2	BTDI_TC04	MLV1	Invalid driverObject passed to AddDevice. FM-MLV1 acted at: mov eax, [ebx+0x28]
BT_E3	BTDI_TC05	MLV1	DeviceExtension = NULL passed to AddDevice. FM-MLV1 acted at: mov eax,[ebx+0x28]
BT_E4	BTDI_TC06	MLV1	Dimension of DeviceExtension lower than expected passed to AddDevice FM-MLV1 validator at memset function detected
BT_E5	BTDI_TC08	MLV1	Invalid parameter passed to IRP_MJ_POWER MLV1 acted at: mov eax, [ebx+0x28]
BT_E6	BTDI_TC12	MLV1	Invalid parameter passed to IRP_MJ_CLOSE FM-MLV1 act at: mov eax,[eax+0x8]
BT_E7	BTDI_TC14	MLV1	Invalid parameter passed to IRP_MJ_INTERNAL_DEV_CONTROL MLV1 acted at: mov eax,[eax+0x8]
BT_E8	BTDI_TC16	MLV1	Invalid parameter passed to IRP_MJ_SYSTEM_CONTROL MLV1 acted at: mov eax,[eax+0x8]

The case signalled by error BT_E3 is a slight different variation from the above 2 errors, and occurs because the Test Manager passed a NULL value in the DeviceExtension field of the DeviceObject structure. In this case when the DD tried to access a DeviceExtension field (DeviceExtension was based by the ebx register) in the mov eax,[ebx+0x28] instruction the MLV1-SourceOperand validator triggered the error.

The error BT_E4 shows a situation where the dimension allocated by the OSE to the DeviceExtension was deliberately less than what was assigned by the DUT. This resulted in a buffer overflow caught by the FLV1-MemoryRange when it checked that the final byte of memset was out of the range of the assigned memory to the DUT.

Errors BT_E5 to BT_E8 were all caused by invalid parameter passed to IRP_MJ_POWER, IRP_MJ_CLOSE, IRP_MJ_INTERNAL_DEV_CONTROL and IRP_MJM_SYSTEM_CONTROL respectively. The invalid parameter consisted in filling in the input parameters with the expected parameters for DriverEntry. Whenever these functions try to access the parameter a failure occurs. Curiously, all of these

3 errors were triggered by the same type of instruction `mov eax,[eax+0x8]` although located at different addresses.

Even though errors BT_E1 to BT_E8 can be considered as false positives (these errors are not related with an incorrect implementation from the DUT, but caused by an incorrect assignment of parameters from the OS), it demonstrates how dependent DD are from the kernel. Curiously, under the same initial conditions, function `DriverUnload` did not caused any fault. A deeper analysis to the `DriverUnload` function code revealed that the reason for this is the fact that this function only has a `ret` instruction. Despite of the invalid parameters, since no code tries to access it, no error is signalled.

Finally, Table 7-16 shows the relationship between the tests sets and the identified errors.

Table 7-16: Relation between the test sets and the identified errors.

Test Set	Error ID							
	BT_E1	**BT_E2**	**BT_E3**	**BT_E4**	**BT_E5**	**BT_E6**	**BT_E7**	**BT_E8**
BT_TS1	✓							
BT_TS2		✓	✓	✓				
BT_TS3					✓			
BT_TS4							✓	
BT_TS5								✓
BT_TS6						✓		

Experiments with Serial over Bluetooth Driver

The second set of experiments targeted the `oxser.sys` (SR) serial over Bluetooth DD which is supplied as part of the installation package of the BlueSoleil Bluetooth dongle. Table 7-17 lists the characteristics of the SR DD and contain some statistical data obtained after loading this DUT into the Discovery platform.

The size in disk of the SR DD is 49,408B which are translated into 13,754 TDiscovery memory cells. The cells store the code instructions found in `.text`, `PAGESPR0`, `PAGESRP0` and `PAGESER` sections. During the loading process of this DD, it was found a total of 77 imported functions. Most of them, 68, are imported from `ntoskrnl.exe`. A total of 7 functions are imported from `hal.dll`, and 2 are imported from `wmilib.sys`.

Discovery initially detected 169 internal functions which evolved to 180 resulting from the experiments.

Table 7-17: Characteristics of the SR DD.

Element	Value
ID	SR
Device driver file name	oxser.sys
Type	Serial Bluetooth Emulator
Vendor	IVT Corporation
Target OS	Windows 7
Target Platform	32 bit
Disk Size	49,408B
.text section	Start: 0x6F30380, Size: 9,216B
.rdata section	Start: 0x6F32780, Size: 640B
.data section	Start: 0x6F32A00, Size: 384B
PAGESPR0	Start: 0x6F32B80, Size: 896B
PAGESRP0	Start: 0x6F32F00, Size: 12,800B
PAGESER	Start: 0x6F36100, Size: 15,360B
INIT section	Start: 0x6F39D00, Size: 3,328B
.rsrc section	Start: 0x6F3AA00, Size: 4,224B
.reloc section	Start: 0x6F3BA80, Size: 1,684B
DriverEntry address	0x6F39D00
Number of TDiscovery memory cells	13,754
Number of imported functions (ntoskrnl.exe)	77
Number of imported functions (hal.dll)	7
Number of imported functions (wmilib.sys)	2
Initial number of local DD functions	169
Final number of local DD functions	180

Imported Functions Test Cases for SR

Table 7-18 represents the eligible imported functions used by the SR DD and the corresponding test cases. Similarly, to what happened to BT DD, it was discarded from the tests all the imported functions that do not have return values and output parameters.

Additionally, imported functions that potentially could lead to false positives were not considered as well. IFTC6 was also not considered as part of the test cases because it could lead to false positive results (represented in the table by a shaded area in IFTC6 column).

Table 7-18: SR imported functions test cases.

SR Imported Functions	Test Cases								
	IFTC1	IFTC2	IFTC3	IFTC4	IFTC5	IFTC6	IFTC7	IFTC8	IFTC9
ExAllocatePoolWithQuotaTag	✓	✓				-			
ExAllocatePoolWithTag	✓	✓				-			
IoAllocateErrorLogEntry	✓	✓				-			
IoAttachDeviceToDeviceStack	✓	✓				-			
IoBuildSynchoronousFsdRequest				✓		-	✓		
IoCancelIrp	✓	✓				-			
IoConnectInterrupt				✓		-	✓		
IoCreateDevice				✓		-	✓		
IoCreateSymbolicLink	✓	✓				-			
IofCallDriver				✓		-	✓		
IoGetConfigurationInformation	✓	✓				-			
IoOpenDeviceRegistryKey				✓		-	✓		
IoRegisterDeviceInterface				✓		-	✓		
IoSetDeviceInterfaceState	✓	✓				-			
IoWMRegistrationControl	✓	✓				-			
KeCancelTimer				✓		-	✓		
KeInsertQueueDpc				✓		-	✓		
KeRemoveQueueDpc				✓		-	✓		
KeSynchronizeExecution				✓		-	✓		
KeWaitForSingleObject	✓	✓				-			
PoCallDriver				✓		-	✓		
PoRequestPowerIrp				✓		-	✓		
RtlDeleteRegistryValue	✓	✓				-			
RtlIniUnicodeString						-		✓	✓
RtlIntegerToUnicodeString				✓		-	✓		
RtlQueryRegistryValues			✓	✓	✓	-	✓		
WmiCompleteRequest				✓		-	✓		
WmiSystemControl				✓		-	✓		
ZwClose	✓	✓				-			
ZwQueryValueKey				✓		-	✓		
ZwSetValueKey				✓		-	✓		

✓ Test condition

Test Cases for SR at Driver Interface

Table 7-19 and Table 7-20, presents the SR DD interface functions directly tested by Discovery. These functions were automatically detected after the execution of the `DriverEntry` function.

Table 7-19: SR Driver Interface and test cases.

Target Interface Function	Test Case ID	Test Case Description
DriverEntry	SRDI_TC01	Valid parameters
AddDevice	SRDI_TC02	Valid parameters
DriverUnload	SRDI_TC03	Valid parameters
IRP_MJ_CLEANUP	SRDI_TC04	Valid request (no input/output parameters exist for this request).
IRP_MJ_CLOSE	SRDI_TC05	Valid request (no input/output parameters exist for this request).
IRP_MJ_CREATE	SRDI_TC06	stackLocation.MajorFunction = IRP_MJ_CREATE
	SRDI_TC07	Invalid stackLocation.MajorFunction value
IRP_MJ_DEVICE_CONTROL	SRDI_TC08 to SRDI_TC14	stackLocation.MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL stackLocation.Parameters.DeviceControl.InputBufferLength = 0x1 stackLocation.Parameters.DeviceIoControl.IoControlCode = collection of values determined from the `cmp` instructions
	SRDI_TC15	Invalid stackLocation.MajorFunction
IRP_MJ_FLUSH_BUFFERS	SRDI_TC16	stackLocation.MajorFunction = IRP_MJ_CREATE
	SRDI_TC17	Invalid stackLocation.MajorFunction value
IRP_MJ_INTERNAL_DEVICE_CONTROL	SRDI_TC18 to SRDI_TC24	stackLocation.MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL stackLocation.Parameters.DeviceControl.InputBufferLength = 0x1 stackLocation.Parameters.DeviceIoControl.IoControlCode = collection of values determined from the cmp instructions
	SRDI_TC25	Invalid stackLocation.MajorFunction
IRP_MJ_POWER	SRDI_TC26	stackLocation.MajorFunction = IRP_MJ_POWER stackLocation.MinorFunction = IRP_MN_POWER_SEQUENCE
	SRDI_TC27	stackLocation.MajorFunction = IRP_MJ_POWER stackLocation.MinorFunction = IRP_MN_QUERY_POWER
	SRDI_TC28	stackLocation.MajorFunction = IRP_MJ_POWER stackLocation.MinorFunction = IRP_MN_SET_POWER
	SRDI_TC29	stackLocation.MajorFunction = IRP_MJ_POWER stackLocation.MinorFunction = IRP_MN_WAIT_WAKE
	SRDI_TC30	stackLocation.MajorFunction = IRP_MJ_POWER stackLocation.MinorFunction = invalid Value

Table 7-20: SR Driver Interface and test cases (continued).

Target Interface Function	Test Case ID	Test Case Description
IRP_MJ_QUERY_INFORMATION	SRDI_TC31	stackLocation.MajorFunction = IRP_MJ_QUERY_INFORMATION IrpSp->Parameters. QueryFile.FileInformationClass = FileAllInformation
	SRDI_TC32	Invalid IrpSp->Parameters. QueryFile.FileInformationClass
IRP_MJ_READ	SRDI_TC33	stackLocation.MajorFunction = IRP_MJ_READ IrpSp->MinorFunction = IRP_MN_NORMAL IrpSp->Parameters.Read.Length =0
	SRDI_TC34	stackLocation.MajorFunction = IRP_MJ_READ IrpSp->MinorFunction = IRP_MN_NORMAL IrpSp->Parameters.Read.Length =0xA
	SRDI_TC35	stackLocation.MajorFunction = IRP_MJ_READ IrpSp->MinorFunction = IRP_MN_NORMAL IrpSp->Parameters.Read.Length =0xFF
IRP_MJ_SET_INFORMATION	SRDI_TC36	IrpSp->MajorFunction = IRP_MJ_SET_INFORMATION IrpSp-> Parameters.SetFile. FileInformationClass = FileBasicInformation
	SRDI_TC37	IrpSp->MajorFunction = IRP_MJ_SET_INFORMATION IrpSp-> Parameters.SetFile. FileInformationClass = invalid value
IRP_MJ_SYSTEM_CONTROL	SRDI_TC38	IrpSp->MajorFunction = IRP_MJ_SYSTEM_CONTROL IrpSp->MinorFunction = IRP_MN_ENABLE_EVENTS
	SRDI_TC39	IrpSp->MajorFunction = IRP_MJ_SYSTEM_CONTROL IrpSp->MinorFunction = invalid value
IRP_MJ_WRITE	SRDI_TC40	Irp->AssociatedIrp.SystemBuffer uses buffered I/O Parameters.Write.Length = 0x00
	SRDI_TC41	Irp->AssociatedIrp.SystemBuffer uses buffered I/O Parameters.Write.Length = 0x0A
	SRDI_TC42	Irp->AssociatedIrp.SystemBuffer uses buffered I/O Parameters.Write.Length = 0xFFFF

To reduce the number of test cases and the number of false positives, no invalid parameters have been used for `DriverEntry`, `AddDevice` and `DriverUnload` function (SRDI_TC01 to SRDI_TC03). As demonstrated by the tests performed in the BT DD, passing invalid parameters to these functions causes false positive results.

The IRP_MJ_CLEANUP and IRP_MJ_CLOSE dispatch functions have no input/output parameters. Therefore, no special conditions are used to analyse them (SRDI_TC04 and SRDI_TC05).

On the contrary, IRP_MJ_CREATE receives an input value in the stackLocation.MajorFunction member of the IRP request. Two situations are evaluated, stackLocation.MajorFunction equal to IRP_MJ_CREATE (SRDI_TC06) and stackLocation.MajorFunction equal to an unexpected value (SRDI_TC07).

The IRP_MJ_DEVICE_CONTROL is analysed with test cases SRDI_TC08 to SRDI_TC14. In these test cases, Discovery employs the values used in cmp instructions as candidates for the IoControlCode passed as parameter. The idea is find out which IoControlCode this dispatch function is using. SRDI_TC15 tests the condition of having an invalid MajorFunction passed to IRP_MJ_DEVICE_CONTROL.

The values used to analyse the handling of the remaining IRP_MJ_XXXX dispatch functions, follows the same logic as the previous test conditions (i.e., exercising the dispatch functions with meaningful parameters taking into consideration the input parameter types and possible values).

Test Set for SR

Table 7-21 defines 191 test cases grouped into 15 test sets, where the SR DD code is analysed through different function call combinations. These test sets represent possible calling sequences during the SR execution in the OS. The test cases used to analyse each function in each call sequence result from the tests identified for the interface function of the DD and the test cases identified for the used imported functions.

Table 7-21: Test Set for SR.

Test Set	Call Sequence	Total Test Cases	Execution Time (s)	
			Partial	Total
SR_TS01	DriverEntry	8	165,6	167,8
	DriverUnload	1	2,2	
SR_TS02	DriverEntry	1	20,7	31,7
	AddDevice	4	8,8	
	DriverUnload	1	2,2	
SR_TS03	DriverEntry	1	20,7	24,7
	AddDevice	1	2,2	
	IRP_MJ_CLEANUP	1	1,8	
SR_TS04	DriverEntry	1	20,7	24,9
	AddDevice	1	2,2	
	IRP_MJ_CLOSE	1	2,0	

Table 7-22: Test Set for SR (continued).

Test Set	Call Sequence	Total Test Cases	Execution Time (s)	
			Partial	Total
SR_TS05	DriverEntry	1	20,7	159,2
	AddDevice	1	2,2	
	IRP_MJ_CREATE	64	136,3	
SR_TS06	DriverEntry	1	20,7	83,7
	AddDevice	1	2,2	
	IRP_MJ_DEVICE_CONTROL	32	60,8	
SR_TS07	DriverEntry	1	20,7	26,4
	AddDevice	1	2,2	
	IRP_MJ_FLUSH_BUFFERS	2	3,5	
SR_TS08	DriverEntry	1	20,7	47,2
	AddDevice	1	2,2	
	IRP_MJ_INTERNAL_DEV_CONTROL	10	24,3	
SR_TS09	DriverEntry	1	20,7	55,5
	AddDevice	1	2,2	
	IRP_MJ_POWER	16	32,6	
SR_TS10	DriverEntry	1	20,7	31,4
	AddDevice	1	2,2	
	IRP_QUERY_INFORMATION	6	8,5	
SR_TS11	DriverEntry	1	20,7	31,3
	AddDevice	1	2,2	
	IRP_MJ_READ	4	8,4	
SR_TS12	DriverEntry	1	20,7	24,8
	AddDevice	1	2,2	
	IRP_MJ_SET_INFORMATION	1	1,9	
SR_TS13	DriverEntry	1	20,7	26,9
	AddDevice	1	2,2	
	IRP_MJ_SYSTEM_CONTROL	2	4,0	
SR_TS14	DriverEntry	1	20,7	31,7
	AddDevice	1	2,2	
	IRP_MJ_WRITE	4	8,8	
SR_TS15	DriverEntry	1	20,7	37,2
	AddDevice	1	2,2	
	IRP_MJ_CREATE	1	2,1	
	IRP_MJ_DEVICE_CONTROL	1	1,9	
	IRP_MJ_READ	1	2,1	
	IRP_MJ_WRITE	1	2,2	
	IRP_MJ_CLEANUP	1	1,8	
	IRP_MJ_CLOSE	1	2,0	
	DriverUnload	1	2,2	

Test Results for SR

The execution of the SR driver code, resulted in the detection of the errors summarized at Table 7-23.

The SR_E01 error occurs in all test sets where `DriverEntry` and `DriverUnload` have successful executions and are called in sequence. This order of events (detected in SR_TS01, SR_TS02 and SR_TS15) triggers an error caught by the FM-PEV1 Resource Leakage Validator.

Table 7-23: Test Results for SR.

Error ID	Test Set ID	Failure Mode	Error description
SR_E01	SR_TS02 SR_TS15	PEV1	Memory leakage.
SR_E02	SR_TS02 to SR_TS15	FLV6	The return code of RtlIntegerToUnicodeString is not validated.
SR_E03	SR_TS01	PEV2	Dormant path at Driver Entry.
SR_E04	SR_TS01	FLV7	Explicit call to KbdBugCheck when BreakOnEntry registry exists with value different from zero.

This occurs because a portion of 30 bytes of memory allocated with function `ExAllocatePoolWithTag` in the `AddDevice` function is not returned to the OSE by the `DriverUnload` function. Although the leakage is small, and requires the activation/deactivation of the DD (which under normal situations is not usual to happen often), it may be exploited to crash the system.

The situation reported in SR_E02 is triggered by FM-FLV6 because the DD does not validate the return value of `RtlIntegerToUnicodeString` when is called by `InternalFunction_0117` (which in turn is called by `AddDevice`). The `RtlIntegerToUnicodeString` function is used by the SR DD to build the device name passed to function `IoCreateDevice` (called in InternalFunction_0117). Although in most situations it is not expected that the `RtlIntegerToUnicodeString` returns an error, if it ever does, it may be impossible for applications to connect with this DD to perform I/O requests.

During the analysis to the `DriverEntry` function, a call to `RtlQueryRegistryValues` is performed to obtain the values of specific SR Windows Registry Keys: `BreakOnEntry`, `DebugLevel`, `ForceFiFoEnable`, `RxFIFO`, `TxFIFO`, `PermitShare` and `LogFifo`. When the Test Manager forced the return of IFTC3 values on function `RtlQueryRegistryValues`, the PEV2 Dormant Code Validator raised an error. When the Test Manager forced IFTC5 values returned by function `RtlQueryRegistryValues` the dormant code was activated and a call to the OSE function `KbdBugCheck` is performed which triggers FLV7. In fact, the SR_E04 error represents a vulnerability to all the systems that have the SR DD installed. By placing a `BreakOnEntry` key with value `0x01` into the Windows registry path of this DD, it is possible to cause a crash whenever the system boots and SR is activated. Once this vulnerability is triggered, this situation can only be reverted either by using the secure mode and recover the last good known configuration of Windows, by booting with another image disk or reinstalling the OS.

Performance of Discovery

This section is dedicated to a brief analysis over the performance of Discovery. We are going to take as examples for the analysis the BT and SR DD used in the previous sections.

Table 7-24 shows the execution time of Discovery related with the DUT loading and initialization of the execution platform.

Table 7-24: Execution time of Discovery during the loading process.

Metric	BT	SR
File size	20,480B	51,169B
Number of imported functions	32	77
Initialization of internal structures*	24ms	
File loading	16ms	16ms
Linkage	284ms	375ms
Sections processing	132ms	517ms
Platform initialization*	169ms	
Total time	625ms	1,077ms

*These values are intrinsic to the platform and independent from the DUT

Considering that the time to initialize the internal structures and the platform of Discovery is independent from the DUT, the overall time to load the driver and be ready to start the analysis is influenced by the complexity of the linkage process. Since SR is more complex (inferred by the number of imported functions and lines of code), Discovery takes more time to perform the loading process (1,077ms) than for BT (625ms). However, both DUT are ready for analysis in the order of less than 1 second.

During the analysis process, Discovery takes the values listed in Table 7-25 to emulate various instructions. The table represents a sequence execution sample of 278 instructions, which took a total of 5,111ms. The first column of the table has the instruction mnemonic, the second column the minimum time spent for the instruction execution, followed by the average execution time and finally, the last column, has the maximum value observed for the instruction execution.

From the sample, it can be observed that the `ret` instructions has the highest execution time, which has to do with the context switching that occurs from the true-mode to the emulation-mode. The `push` instruction has the second highest time with an average execution time of 54ms. The remaining instructions are executed in average in less than 10ms. The `mov` instructions takes in average 2.9ms.

Comparing the performance of the execution platform of Discovery with a modern CPU (for instance an Intel Core i7 performs 0.318MIPms @ 3.0GHz [171]), one can conclude that there is plenty for improvement.

Table 7-25: Performance of Discovery during execution.

Instruction	Time (ms)**		
	Min	Average	Max
ret	80	93.3	113
push	1	54.3	57
jnz	2	6.0	21
lea	2	5.4	10
call	2	3.7	5
pop	2	3.2	4
sub	2	3.0	4
and	3	3.0	3
mov	1	2.9	8
xor	2	2.9	4
jz	2	2.5	3
test	2	2.3	3
cmp	2	2.3	3
stosd	2	2.2	3
leave	2	2.0	2
not	2	2.0	2
dir*	1	1.7	3
jmp	1	1.7	2
add	1	1.0	1

*This instruction does not exist in a real x86-64 platform. It is an abstraction to direct assign a value to a register used during context switching.
**Sample execution involving a sequence of 278 instructions.

However, the primary goal of Discovery was not performance, and from our knowledge of the platform, the average times can be significantly reduced at least by a factor of 10. Nevertheless, for some of the experiments performed, some Validators have been triggered after a few instructions, which is to say that Discovery could detect errors in a few milliseconds time.

Another important aspect of the Discovery platform is the ability to automatically maintain the execution context to speed up testing. Taking a closer look to the last column of Table 7-13 and Table 7-21, all the time spent with `DriverEntry` and `AddDevice` can be avoided, which significantly reduces the overall time of the analysis.

7.11 Summary

The Supervised Emulation Analysis is a methodology for the detection and location of flaws in DD without resorting to the source code or specific hardware. The methodology was designed based on: i) the assumption that DD follow a specific driver model structure which limits the interface from which the OS and the DD interact with each other; ii) the types of bug classes that can be detected and located based in the type of the executing hardware platform; iii) the definition of validators that locate the considered bug classes; iv) the definition of an emulation platform that analyses the DD binary code and v) the necessary procedures that should be in place to locate DD flaws.

The driver model establishes the internal structure of the DD. The entry point is the only known function by the OS immediately after the DD being loaded. During the DD execution, the DD registers limited interface functions in the OS that implement specific services required by the OS. The DD may have many internal functions used to simplify its code organization. These functions are used by some of the interface functions exposed to the OS and other internal functions. However, the OS cannot interface directly with them. DD may register interrupt functions in the OS, but these functions also follows a specific model. The DD depends on functions typically provided by the OS that form the API that the DD can use.

The DD binary file follows a specific format which can be interpreted to determine and locate its various components which are fundamental to load and be able to execute the DD code.

Based on the previous information it should be possible to build a system that can interface the DD code and perform the same tasks as the OS, testing the DD through the various interface functions and locate errors by using test cases that address the parameters and return values of the interface.

The methodology defines validators. A mechanism called during the DD code execution to perform a check over an intended action. Three different kind of validators classes were identified: i) Machine Level Validators that are triggered during the execution of a machine instruction to check the validity of the machine instruction parameters; ii) Function Level Validators that are triggered during the execution of a call from the DD to the OS and iii) Post Execution Validators that are triggered after the execution of a sequence of DD interface functions to detect abnormal situations such as resource leakage and dormant code.

The methodology defines an emulated environment where the DD code is loaded and executed. The execution of each DD machine instruction is subject to the action of the validators to ascertain the correctness of the execution. The emulation ensures that there is no need for the hardware of the platform or device during the DD analysis. Additionally, the stability of the testing platform is not compromised by the tests being performed because the errors in the DD code are detected before the execution can take place. Finally, the identification of the flaws can be distributed over different systems.

The procedures for detecting and locating the flaws in the DD consists of: i) a preparation phase where the DD binary file is loaded in the emulation platform; ii) pre-processing of the DD to identify the code structure; iii) exercise the DD using a set of calling sequences and test cases that mimic the OS behaviour but create typical and extraordinary scenarios that the DD should handle.

Discovery is an implementation of the Supervised Emulation Analysis methodology. It implements an emulation of the x86-64 architecture platform where the DD code analysis occurs, an Operating System Emulator to load and interface the DD code, a Device Emulator structure, a Database to handle configurations, traces, test sequences, test cases and results, and the Test Manager in charge of supervising the strategy employed to find the errors in the DD code. Discovery has granularity control over the machine code execution of the DD supporting very detailed checks at the level of each machine instruction execution, allowing for catching platform dependent flaws such as buffer overflows, incorrect pointers, invalid jumps and calls. All functions imported by the DD are emulated by the platform. Checks embedded at each imported function allows the detection of flaws at a higher execution level such as incorrect OS object handlers, pointers and function calls. Post execution checks allows for the detection of resource leakages and dormant code.

Experiments performed with two commercial DD demonstrated the dependency level that DD have from the correctness of the calls made from the OS, resource leakage, non-validation of return values, the presence of dormant code and vulnerable situations related with Windows registery values.

Although the primary objective of Discovery was not performance, from the experiments performed it presented results in a reasonable time frame.

CHAPTER 8 CONCLUSIONS AND FUTURE WORK

This chapter summarizes the main contributions of the work and provides some indications of future research directions.

8.1 Conclusions

The thesis describes several methodologies applied to the discovery of errors in DDs and their causes. The first contribution focus on robustness testing of the functions provided by Microsoft's DD development kit (DDK). In the context of this work, we designed a system that automatically writes the source code of potentially faulty DDs, installs them in the OS, triggers the faults, collects and analyses the results. The execution of each DD, the call of the DDK function with potential erroneous parameters, and consequently the behaviour of the system gives us an idea of how well the OS would cope with the triggered faults. The analysis of the results shows that most targeted functions were unable to offer a protection to the incorrect parameters. A small number of hangs and a reasonable number of crashes were observed, which suggests a deficient error containment capability of these OS functions.

The second contribution of this work uses the fuzzing concept to perform the injection of attacks on Wi-Fi DDs. We developed a methodology and an architecture capable of injecting Wi-Fi frames with controlled faulty values in the various frame fields of this medium. A target windows smartphone device is connected to both the Wi-Fi medium and to a host computer that monitors the results of the attacks. The results demonstrated that in most cases, Windows was able to handle correctly the malicious frames. However, the results also showed that Wdev-Fuzzer can be successfully applied to reproduce denial of service attacks using Disassociation and Deauthentication frames. The system revealed a potential implementation problem of the TCP-IP stack, uncovered by the use of disassociation frames when the target device was associated and authenticated with a Wi-Fi access point. The experiments also discovered a previously unknown vulnerability that causes OS hangs, when a specific value was assigned to the TIM element in the Beacon frame.

Another contribution of the work resulted in the Intercept tool that instruments Windows DDs by logging the driver interactions with the OS at function level. It uses an approach where the DD binary is in full control of a DD wrapper layer and the execution is traced to a file recording all function calls, parameter and return values. The trace is directly generated in clear text with all the involved data structures. Intercept gives a clear picture of the dynamics of the driver, which can help in debugging and reverse engineering processes with low performance degradation. Results show the ability of the tool to identify bugs in drivers, by executing tests based on the knowledge obtained from the driver's dynamics.

The final contribution of the work is the Supervised Emulation Analysis methodology and the Discovery framework. The methodology takes advantage of the fact that DD have a well-defined structure, therefore limiting the number of possible path combinations per function and loops. The methodology uses emulation to exercise the DD through its interfaces, mimicking the OS behaviour and verifying the driver execution. The emulation platform controls the binary execution of the DD code with instruction granularity, which enables fine grain checks with Validators that ascertain the validity of the code being executed, this way enabling the detection of low level errors. The emulation platform also provides all the resources required by the DD. Therefore, the platform can catch function level errors related with parameter values, DD state, function call orders and resource leakage. Post Execution Validators can be used to verify the balance of resources and dormant code. Experimental results with Discovery confirmed that the DDs have a high dependency from the OS and do not check (and in most the cases have no way to check) the validity of the parameters passed by the OS, either when calling a DD function or when a OS service returns. The results also show that most the

tested DDs verifies the return values of the OS functions and act accordingly. Nevertheless, it was possible to detect cases where the DDs do not validate return values, present resource leakages and dormant code that may compromise system stability.

8.2 Future Work

Future works can naturally continue to improve and expand the functionalities of the presented tools, methodologies and frameworks to support the detection of DD errors and build more dependable computer systems. Next we present a few ideas for future work within the same research field.

Emulation sandboxing for runtime protection

The thesis addressed the detection of DD errors using an emulated platform to stimulate the DD code while facing specific input values. The operation is performed in an emulated environment. This idea could be extended to an active real-time detection and protection mechanism by creating an emulated sandboxing environment for runtime protection that validates the DD code path before it is executed. In the case of an error being detected, the sandbox can gracefully return to the OS.

Binary code refactoring for error detection on OS resource usage

OS provide resources to DDs. An incorrect use of such resources can lead to leakages and deadlocks. One potential research area that can expand the possibilities of Discovery involves code refactoring of the DD aiming for a fast and accurate detection on errors related with OS resource usage. The main idea is to identify the code paths that involve the allocation/deallocation of resources, acquisition and release of locking mechanisms (locks, semaphores and mutex) and strip out the remaining code. This way, it would be possible to continue to have the underlying usage logic of such resources striped out from the complexity of the other code.

Emulation assisted symbolic execution

An emulation execution platform, such as Discovery, enables the control of the execution engine. As seen during the experimental results of the Supervised Emulation Analysis, DD are highly dependent on the OS inputs. One possible way to achieve higher levels of independency of the OS is to execute each machine

instructions abstractly. Instead of executing the associated algorithm of each machine instruction, the execution engine can be changed to symbolically process the instructions and simplify the correlation between decision instructions, the input parameters of the function and potential errors.

ANNEX I – Robustness Testing of the Windows DDK sample code

This section relates to Chapter 4 Robustness Testing of the Windows Driver Kit. It contains the source code of the device driver template used by DevBuilder when building synthetic device drivers (see DevInjector.c next).

DevBuilder rewrites the DevInjector.c file by adding specific code next to the following comment lines:

- //INSERT DECLARATIONS HERE;
- //INSERT FUNCTION CALL HERE;
- //INSERT POSTCODE;
- //INSERT DRIVER ENTRY CODE.

The code to be inserted at the comment lines identified earlier is found at the XML file that describes the signature of the function to be tested. The next sections of this annex contain:

- devInjector.c
  The synthetic template device driver source file;
- IoCallDriver.XML
  The signature description file of the function IoCallDriver used to generate specific DD for testing the robustness of this function;
- IoCallDriver_1.c
  The resulting source code of a synthetic driver by processing DevInjector.c and IoCallDriver.xml

DevInjector.c

The following text is the source code of the synthetic template device driver used by DevBuilder to build synthetic device drivers.

```
//--------------------------------------------------------------------
//
// DevInjector.c
//
// Copyright (C) 2017 Manuel Mendonca, Nuno Neves
// FCUL
//
// Template driver.
//
// V3.0 Support for post code
//
//
//--------------------------------------------------------------------
#define _X86_
#include "ntddk.h"
#include "..\DD_Include\ioctlcmd.h"

#define DEBUG_IOCTL_TEXT    "DInject - IOCTL_EXECUTE_ACTION invoked.\r\n"
#define DEBUG_IOCTL_DEFAULT  "DInject - IOCTL_EXECUTE_ACTION default.\r\n"
#define DEBUG_DRIVER_ENTRY "DInject - Driver Entry\r\n"
#define DEBUG_DRIVER_ENTRYEND  "DInject - Driver Entry end.\r\n"
#define DEBUG_UNLOAD                 "DInject - Unload.\r\n"

#define SYMBOL_LINK                  L"\\Device\\DInject"
#define DEVICE_LINK         L"\\DosDevices\\DInject"


//--------------------------------------------------------------------
//
// DevInjectorDeviceControl
//
//--------------------------------------------------------------------
NTSTATUS
DevInjectorDeviceControl(
         IN PFILE_OBJECT FileObject,
         IN BOOLEAN Wait, IN PVOID InputBuffer,
         IN ULONG InputBufferLength,
         OUT PVOID OutputBuffer,
         IN ULONG OutputBufferLength,
         IN ULONG IoControlCode,
         OUT PIO_STATUS_BLOCK IoStatus,
         IN PDEVICE_OBJECT DeviceObject,
         IN PIRP Irp)
{
         //INSERT DECLARATIONS HERE

         //IoStatus->Information = 0;
         //OutputBufferLength = 0;

         switch ( IoControlCode ) {
                  case IOCTL_EXECUTE_ACTION:

                  //INSERT FUNCTION CALL HERE


                  //INSERT POSTCODE

                  DbgPrint(DEBUG_IOCTL_TEXT);
                  DbgPrint(OutputBuffer);
                  IoStatus->Status = STATUS_SUCCESS;
                  break;

             default:
                  IoStatus->Status = STATUS_NOT_SUPPORTED;
                  DbgPrint(DEBUG_IOCTL_DEFAULT);
                  break;
         }
         return IoStatus->Status;
}
```

```
//--------------------------------------------------------------------
//
// DevInjectorDispatch
//
// In this routine requests to our own device. The only
// requests we care about handling explicitely are IOCTL commands that
// we will get from the GUI. We also expect to get Create and Close
// commands when the GUI opens and closes communications with us.
//
//--------------------------------------------------------------------
NTSTATUS DevInjectorDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
        PIO_STACK_LOCATION      iosp;
        PVOID                   inputBuffer;
        PVOID                   outputBuffer;
        ULONG                   inputBufferLength;
        ULONG                   outputBufferLength;
        ULONG                   ioControlCode;
        NTSTATUS                status;
        //
        // Switch on the request type
        //
        iosp = IoGetCurrentIrpStackLocation (Irp);
        switch (iosp->MajorFunction) {

        case IRP_MJ_CREATE:
        case IRP_MJ_CLOSE:
                status = STATUS_SUCCESS;
                break;

        case IRP_MJ_DEVICE_CONTROL:
                inputBuffer       = Irp->AssociatedIrp.SystemBuffer;
                inputBufferLength = iosp->Parameters.DeviceIoControl.InputBufferLength;
                outputBuffer      = Irp->AssociatedIrp.SystemBuffer;
                outputBufferLength = iosp-> Parameters.DeviceIoControl.OutputBufferLength;
                ioControlCode      = iosp-> Parameters.DeviceIoControl.IoControlCode;

               status = DevInjectorDeviceControl(
                        iosp->FileObject,
                        TRUE,inputBuffer, inputBufferLength, outputBuffer,
                        outputBufferLength, ioControlCode, &Irp->IoStatus,
                   DeviceObject, Irp);
                break;

        default:

        status = STATUS_INVALID_DEVICE_REQUEST;
        break;
        }

        //
        // Complete the request
        //
        Irp->IoStatus.Status = status;
        IoCompleteRequest( Irp, IO_NO_INCREMENT );
        return status;
}


//--------------------------------------------------------------------
//
// DevInjectorUnload
//
// Our job is done - time to leave.
//
//--------------------------------------------------------------------
VOID
DevInjectorUnload(IN PDRIVER_OBJECT DriverObject)
{
        WCHAR                   deviceLinkBuffer[]  = SYMBOL_LINK;
        UNICODE_STRING          deviceLinkUnicodeString;

        //
        // Delete the symbolic link for our device
        //
        RtlInitUnicodeString( &deviceLinkUnicodeString, deviceLinkBuffer );
        IoDeleteSymbolicLink( &deviceLinkUnicodeString );
```

```
        //
        // Delete the device object
        //
        IoDeleteDevice( DriverObject->DeviceObject );
        DbgPrint(DEBUG_UNLOAD);
}



//--------------------------------------------------------------------
//
// DriverEntry
//
// Installable driver initialization. Here we just set ourselves up.
//
//--------------------------------------------------------------------
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
        NTSTATUS                status;
        WCHAR                   deviceNameBuffer[]  = SYMBOL_LINK;
        UNICODE_STRING          deviceNameUnicodeString;
        WCHAR                   deviceLinkBuffer[]  = DEVICE_LINK;
        UNICODE_STRING          deviceLinkUnicodeString;
        PDEVICE_OBJECT          interfaceDevice = NULL;
        ULONG                   startType, demandStart;
        RTL_QUERY_REGISTRY_TABLE paramTable[2];
        UNICODE_STRING          registryPath;
        LARGE_INTEGER           crashTime;

        //INSERT DRIVER ENTRY CODE
        DbgPrint(DEBUG_DRIVER_ENTRY);

        //
        // Create a named device object
        //
        RtlInitUnicodeString (&deviceNameUnicodeString,deviceNameBuffer );

        status = IoCreateDevice ( DriverObject,
                               0,
                               &deviceNameUnicodeString,
                               FILE_DEVICE_DEVINJECT,
                               0,
                               TRUE,
                               &interfaceDevice );
        if (NT_SUCCESS(status)) {
                //
                // Create a symbolic link that the GUI can specify to
                // gain access to this driver/device
                //
                RtlInitUnicodeString (&deviceLinkUnicodeString, deviceLinkBuffer);
                status = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                                &deviceNameUnicodeString );

                //
                // Create dispatch points for all routines that must be
                // injected
                //
                DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DevInjectorDispatch;
                DriverObject->DriverUnload = DevInjectorUnload;
        }

        if (!NT_SUCCESS(status)) {
                //
                // Something went wrong, so clean up
                //
                DbgPrint("Something Went Wrong");
                if( interfaceDevice ) {
                        IoDeleteDevice( interfaceDevice );
                }
            }

        //
        // Query our start type to see if we are supposed to monitor starting
        // at boot time
        //
        registryPath.Buffer = ExAllocatePool( PagedPool, RegistryPath->Length +
                sizeof(UNICODE_NULL));

        if(!registryPath.Buffer) {
```

```
            return STATUS_INSUFFICIENT_RESOURCES;
        }

        registryPath.Length = RegistryPath->Length + sizeof(UNICODE_NULL);
        registryPath.MaximumLength = registryPath.Length;

        RtlZeroMemory( registryPath.Buffer, registryPath.Length );
        RtlMoveMemory( registryPath.Buffer,  RegistryPath->Buffer, RegistryPath->Length);

        demandStart = SERVICE_DEMAND_START;
        startType = demandStart;
        RtlZeroMemory( &paramTable[0], sizeof(paramTable));
        paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
        paramTable[0].Name = L"Start";
        paramTable[0].EntryContext = &startType;
        paramTable[0].DefaultType = REG_DWORD;
        paramTable[0].DefaultData = &demandStart;
        paramTable[0].DefaultLength = sizeof(ULONG);

        RtlQueryRegistryValues( RTL_REGISTRY_ABSOLUTE, registryPath.Buffer, &paramTable[0],
                NULL, NULL);
        DbgPrint(DEBUG_DRIVER_ENTRYEND);
        return status;
}
```

IoCallDriver.XML

The following text is the XML signature definition of the IoCallDriver function. It will be used by DevBuilder to write the source code of the multiple synthetic device drivers used to perform the robustness test campaign of the IoCallDriver function.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<functions xmlns="www.fcul.pt">
        <function functionName="IoCallDriver">
                <returnValue>NTSTATUS</returnValue>
                <preCode codeLines="PRECODE">
                </preCode>

                <postCode codeLines="POSTCODE">
                </postCode>

                <parameter parameterName="PDEVICE_OBJECT">
                        <value></value>
                        <value>NULL</value>
                        <value>DeviceObject</value>
                </parameter>

                <parameter parameterName="PIRP">
                        <value></value>
                        <value>NULL</value>
                        <value>Irp</value>
                </parameter>

        </function>
</functions>
```

IoCallDriver0.c

The following text is the source code of the first synthetic device used at the robustness testing campaign of the IoCallDriver function.

```
//-------------------------------------------------------------------
//
// IoCallDriver0.c – rewritten from devInject.c
```

```
//
// Copyright (C) 2017 Manuel Mendonca, Nuno Neves
// FCUL
//
//---------------------------------------------------------------------
#define _X86_
#include "ntddk.h"
#include "..\DD_Include\ioctlcmd.h"

#define DEBUG_IOCTL_TEXT             "IoCallDriver0 - IOCTL_EXECUTE_ACTION invoked.\r\n"
#define DEBUG_IOCTL_DEFAULT          "IoCallDriver0 - IOCTL_EXECUTE_ACTION default.\r\n"
#define DEBUG_DRIVER_ENTRY           "IoCallDriver0 - Driver Entry\r\n"
#define DEBUG_DRIVER_ENTRYEND   "IoCallDriver0 - Driver Entry end.\r\n"
#define DEBUG_UNLOAD                          "IoCallDriver0 - Unload.\r\n"

#define SYMBOL_LINK                                      L"\\Device\\IoCallDriver0"
#define DEVICE_LINK                                      L"\\DosDevices\\IoCallDriver0"



//---------------------------------------------------------------------
//
// DevInjectorDeviceControl
//
//---------------------------------------------------------------------
NTSTATUS  DevInjectorDeviceControl(
         IN PFILE_OBJECT FileObject,
         IN BOOLEAN Wait,
         IN PVOID InputBuffer,
         IN ULONG InputBufferLength,
         OUT PVOID OutputBuffer,
         IN ULONG OutputBufferLength,
         IN ULONG IoControlCode,
         OUT PIO_STATUS_BLOCK IoStatus,
         IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
         PDEVICE_OBJECT p0;
         PIRP p1;
         //INSERT PRECODE HERE

         switch ( IoControlCode ) {
                  case IOCTL_EXECUTE_ACTION:
                    DbgPrint("IoCallDriver0.c - IOCTL_EXECUTE_ACTION invoked V1.0.\r\n");
                    IoStatus->Status = IoCallDriver( p0,  p1);
                    if (IoStatus->Status == STATUS_PENDING){
                           RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                           STATUS_PENDING.", 50);
                           OutputBufferLength = 50;
                    }
                    else
                    if (IoStatus->Status == STATUS_HANDLE_NOT_CLOSABLE  ){
                           RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                                    STATUS_HANDLE_NOT_CLOSABLE.", 62);
                           OutputBufferLength = 62;
                    }
                    else
                    if (IoStatus->Status == STATUS_INVALID_HANDLE ){
                           RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS.
                           Value is STATUS_INVALID_HANDLE.", 57);
                           OutputBufferLength = 57;
                    }
                    else
                    if (IoStatus->Status == STATUS_ACCESS_DENIED ){
                           RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                             STATUS_ACCESS_DENIED.", 56);
                           OutputBufferLength = 56;
                    }
                    else
                    if (IoStatus->Status == STATUS_INSUFFICIENT_RESOURCES ){
                           RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                             STATUS_INSUFFICIENT_RESOURCES.", 65);
                           OutputBufferLength = 65;
                    }
                   else
                   if (IoStatus->Status == STATUS_ILLEGAL_FLOAT_CONTEXT ){
                           RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                             STATUS_ILLEGAL_FLOAT_CONTEXT.", 64);
                           OutputBufferLength = 64;
                   }
                   else
```

```
                 if (IoStatus->Status == STATUS_SUCCESS){
                         RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                          STATUS_SUCCESS.", 50);
                         OutputBufferLength = 50;
                }
                else
                if (IoStatus->Status == STATUS_ALERTED){
                         RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                          STATUS_ALERTED.", 50);
                         OutputBufferLength = 50;
                }
                else
                if (IoStatus->Status == STATUS_USER_APC){
                         RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                          STATUS_USER_APC.", 51);
                         OutputBufferLength = 51;
                }
                else
                if (IoStatus->Status == STATUS_TIMEOUT ){
                         RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is
                          STATUS_TIMEOUT.", 50);
                         OutputBufferLength = 50;
                }
                else
                {
                         RtlCopyMemory(OutputBuffer, "Return Type is NTSTATUS. Value is not
                           STATUS_SUCCESS.", 54);
                         OutputBufferLength = 54;
                }
                IoStatus->Information = OutputBufferLength;


                //INSERT POSTCODE

                DbgPrint(DEBUG_IOCTL_TEXT);
                DbgPrint(OutputBuffer);
                IoStatus->Status = STATUS_SUCCESS;
                //IoStatus->Information = OutputBufferLength;
                break;


            default:
                IoStatus->Status = STATUS_NOT_SUPPORTED;
                DbgPrint(DEBUG_IOCTL_DEFAULT);
                break;
        }
        return IoStatus->Status;
}


//----------------------------------------------------------------------
//
// DevInjectorDispatch
//
// In this routine requests to our own device. The only
// requests we care about handling explicitely are IOCTL commands that
// we will get from the GUI. We also expect to get Create and Close
// commands when the GUI opens and closes communications with us.
//
//----------------------------------------------------------------------
NTSTATUS DevInjectorDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp )
{
        PIO_STACK_LOCATION      iosp;
        PVOID                   inputBuffer;
        PVOID                   outputBuffer;
        ULONG                   inputBufferLength;
        ULONG                   outputBufferLength;
        ULONG                   ioControlCode;
        NTSTATUS                status;

    //
    // Switch on the request type
    //
        iosp = IoGetCurrentIrpStackLocation (Irp);
        switch (iosp->MajorFunction) {
                case IRP_MJ_CREATE:
                case IRP_MJ_CLOSE:
                        status = STATUS_SUCCESS;
                        break;
                case IRP_MJ_DEVICE_CONTROL:
```

```
                        inputBuffer        = Irp->AssociatedIrp.SystemBuffer;
                        inputBufferLength  = iosp->
                          Parameters.DeviceIoControl.InputBufferLength;
                        outputBuffer       = Irp->AssociatedIrp.SystemBuffer;
                        outputBufferLength = iosp->
                          Parameters.DeviceIoControl.OutputBufferLength;
                        ioControlCode      = iosp->
                          Parameters.DeviceIoControl.IoControlCode;

                        status = DevInjectorDeviceControl( iosp->FileObject, TRUE,
                                        inputBuffer, inputBufferLength,
                                        outputBuffer, outputBufferLength,
                                        ioControlCode, &Irp->IoStatus,
                                        DeviceObject, Irp );
                break;

                default:
                        status = STATUS_INVALID_DEVICE_REQUEST;
                        break;
        }

    //
    // Complete the request
    //
        Irp->IoStatus.Status = status;
        IoCompleteRequest( Irp, IO_NO_INCREMENT );
        return status;
}


//----------------------------------------------------------------------
//
// DevInjectorUnload
//
// Our job is done - time to leave.
//
//----------------------------------------------------------------------
VOID
DevInjectorUnload(IN PDRIVER_OBJECT DriverObject)
{
        WCHAR                   deviceLinkBuffer[]  = SYMBOL_LINK;
        UNICODE_STRING          deviceLinkUnicodeString;

        //
        // Delete the symbolic link for our device
        //
        RtlInitUnicodeString( &deviceLinkUnicodeString, deviceLinkBuffer );
        IoDeleteSymbolicLink( &deviceLinkUnicodeString );

        //
        // Delete the device object
        //
        IoDeleteDevice( DriverObject->DeviceObject );
        DbgPrint(DEBUG_UNLOAD);
}



//----------------------------------------------------------------------
//
// DriverEntry
//
// Installable driver initialization. Here we just set ourselves up.
//
//----------------------------------------------------------------------
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
        NTSTATUS                status;
        WCHAR                   deviceNameBuffer[]  = SYMBOL_LINK;
        UNICODE_STRING          deviceNameUnicodeString;
        WCHAR                   deviceLinkBuffer[]  = DEVICE_LINK;
        UNICODE_STRING          deviceLinkUnicodeString;
        PDEVICE_OBJECT          interfaceDevice = NULL;
        ULONG                   startType, demandStart;
        RTL_QUERY_REGISTRY_TABLE paramTable[2];
        UNICODE_STRING          registryPath;
        LARGE_INTEGER           crashTime;
```

```
        DbgPrint("IoCallDriver0 - DRIVER ENTRY invoked V1.0.\n");
        DbgPrint(DEBUG_DRIVER_ENTRY);

        //
        // Create a named device object
        //
        RtlInitUnicodeString (&deviceNameUnicodeString, deviceNameBuffer);
        status = IoCreateDevice ( DriverObject,
                                  0,
                                  &deviceNameUnicodeString,
                                  FILE_DEVICE_DEVINJECT,
                                  0,
                                  TRUE,
                                  &interfaceDevice);
        if (NT_SUCCESS(status)) {

           //
           // Create a symbolic link that the GUI can specify to gain access
           // to this driver/device
           //
           RtlInitUnicodeString (&deviceLinkUnicodeString,
                                                        deviceLinkBuffer );
           status = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                 &deviceNameUnicodeString );

           //
           // Create dispatch points for all routines that must be injected
           //
           DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]  = DevInjectorDispatch;
           DriverObject->DriverUnload                          = DevInjectorUnload;
        }

        if (!NT_SUCCESS(status)) {
           //
           // Something went wrong, so clean up
           //
           DbgPrint("Something Went Wrong");
           if( interfaceDevice ) {
                 IoDeleteDevice( interfaceDevice );
           }
        }

        //
        // Query our start type to see if we are supposed to monitor starting
        // at boot time
        //
        registryPath.Buffer = ExAllocatePool( PagedPool,
                                    RegistryPath->Length + sizeof(UNICODE_NULL));
        if(!registryPath.Buffer) {

        return STATUS_INSUFFICIENT_RESOURCES;
    }

    registryPath.Length = RegistryPath->Length + sizeof(UNICODE_NULL);
    registryPath.MaximumLength = registryPath.Length;

    RtlZeroMemory( registryPath.Buffer, registryPath.Length );
    RtlMoveMemory( registryPath.Buffer,  RegistryPath->Buffer, RegistryPath->Length  );

    demandStart = SERVICE_DEMAND_START;
    startType = demandStart;
    RtlZeroMemory( &paramTable[0], sizeof(paramTable));
    paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
    paramTable[0].Name = L"Start";
    paramTable[0].EntryContext = &startType;
    paramTable[0].DefaultType = REG_DWORD;
    paramTable[0].DefaultData = &demandStart;
    paramTable[0].DefaultLength = sizeof(ULONG);

    RtlQueryRegistryValues( RTL_REGISTRY_ABSOLUTE, registryPath.Buffer, &paramTable[0],  NULL,
     NULL  );
    DbgPrint(DEBUG_DRIVER_ENTRYEND);
    return status;
}
```

ANNEX II – Discovery

This annex relates to Chapter 7 Supervised Emulation Analysis. It briefly presents Discovery - an application developed to interface with the Discovery framework and locate errors in DD.

Figure AnII-1 depicts the main window of Discovery platform where the DCPU, integrated debugger and main console are shown.

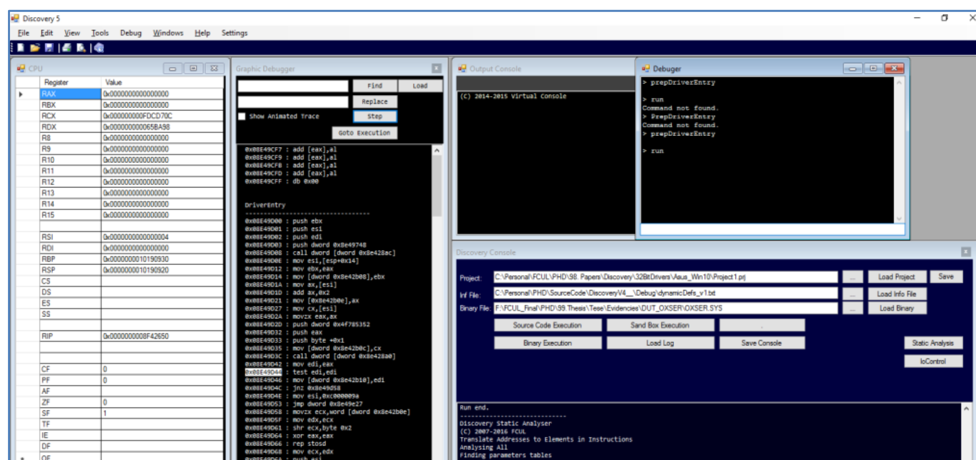


Figure AnII-1: Discovery Main Window (general view).

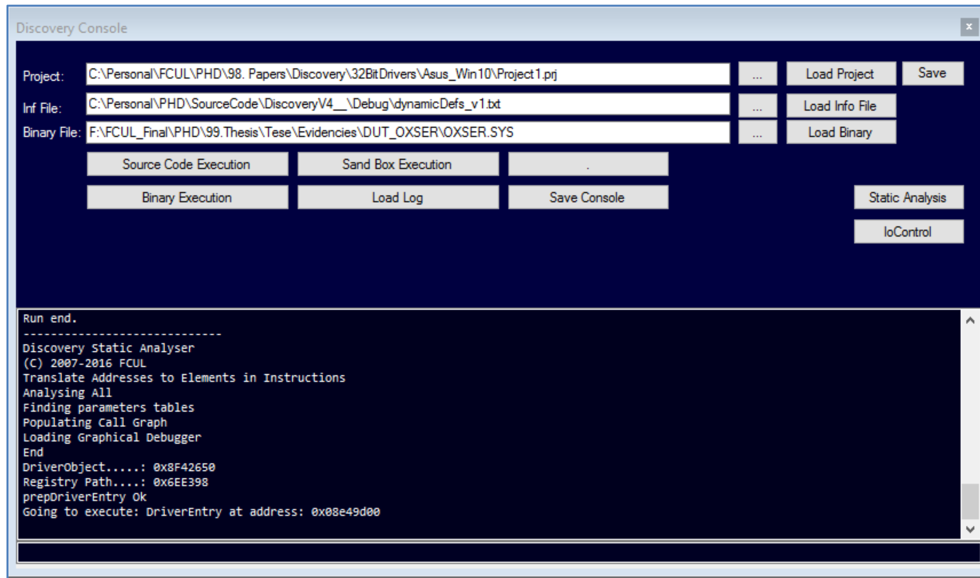


Figure AnII-2: Discovery5 Console.

Figure AnII-2 depicts the console of Discovery through which the user can select the device driver for analysis. The text box labelled “Binary File” contains the path for the device driver under test.

The current version of Discovery allows access to the emulation platform and interaction with the DCPU and integrated debugger (see Figure AnII-3).

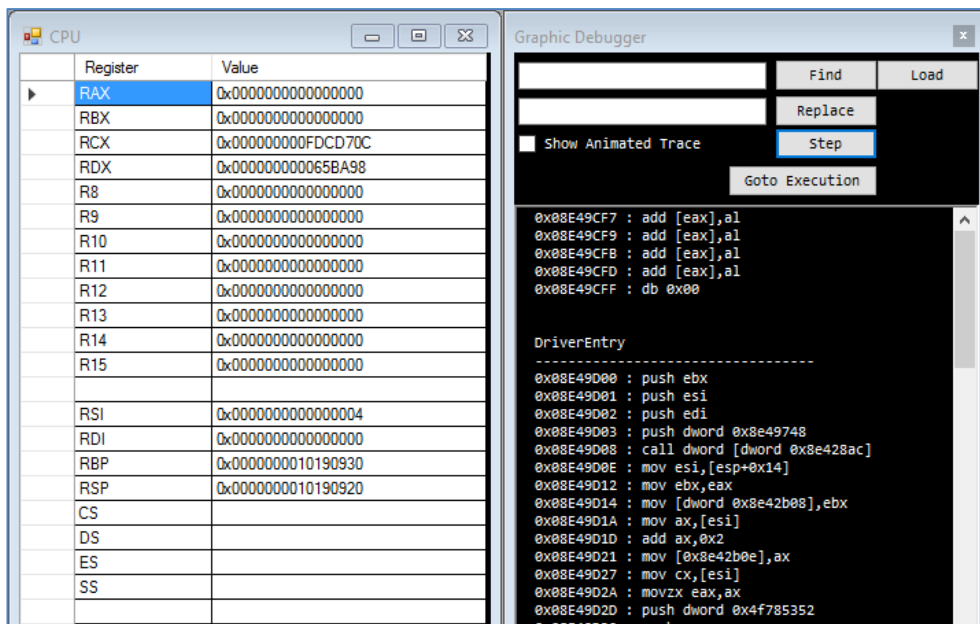


Figure AnII-3: DCPU and integrated debugger windows.

Function Call Graph

Discovery builds the function call graph of the device driver under test. Figure AnII-4 depicts the function call graph available at Discovery with focus on the DriverEntry function of the device driver under test.

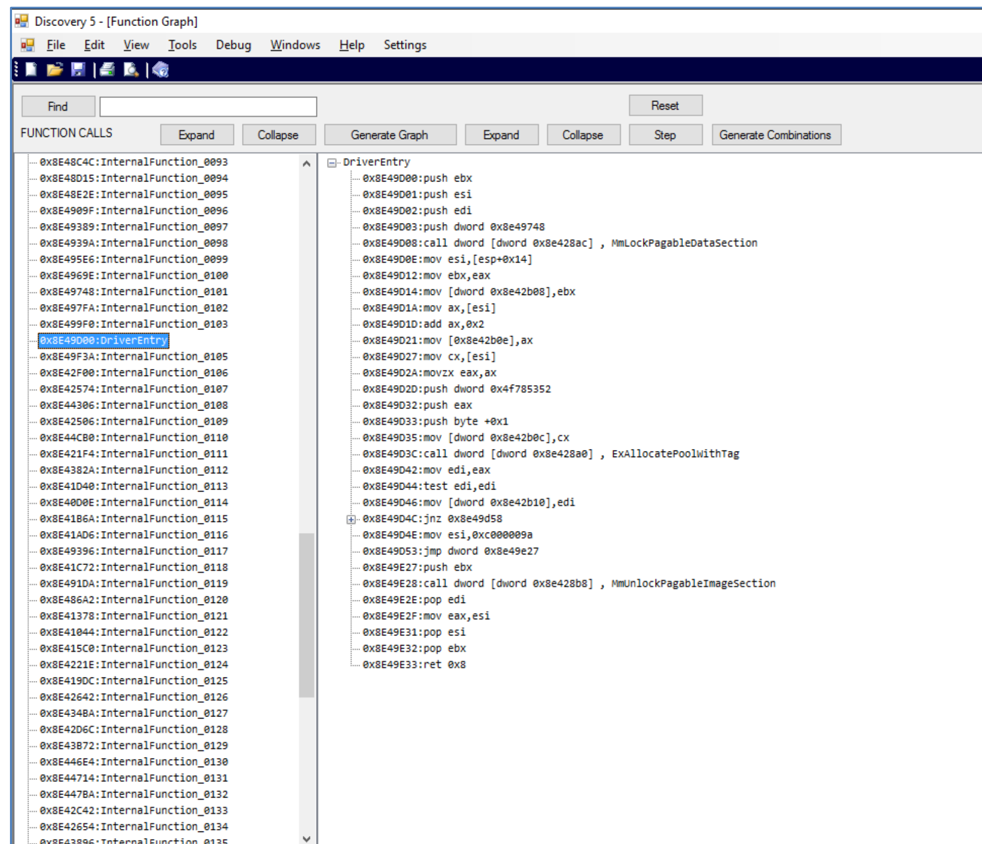


Figure AnII-4: Example of Driver Entry Call Graph (pre-Expanded)

Figure AnII-5 depicts an example of the DriverEntry expanded function call graph. The code highlighted in green shows instructions that where analysed by the platform during the execution of the test sets.

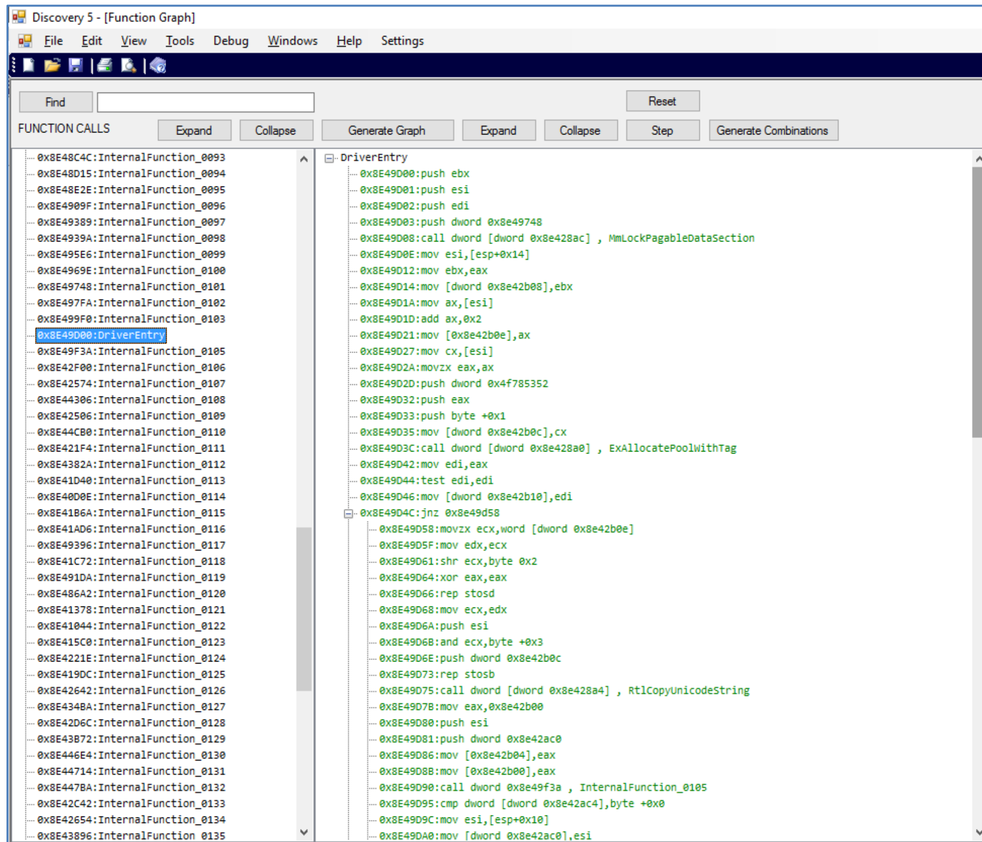


Figure AnII-5: Example of Driver Entry Call Graph (Expanded)

Report

Figure AnII-6 depicts the dynamic report being built as a result of the analysis performed at the device driver and errors being detected by the implemented validators.

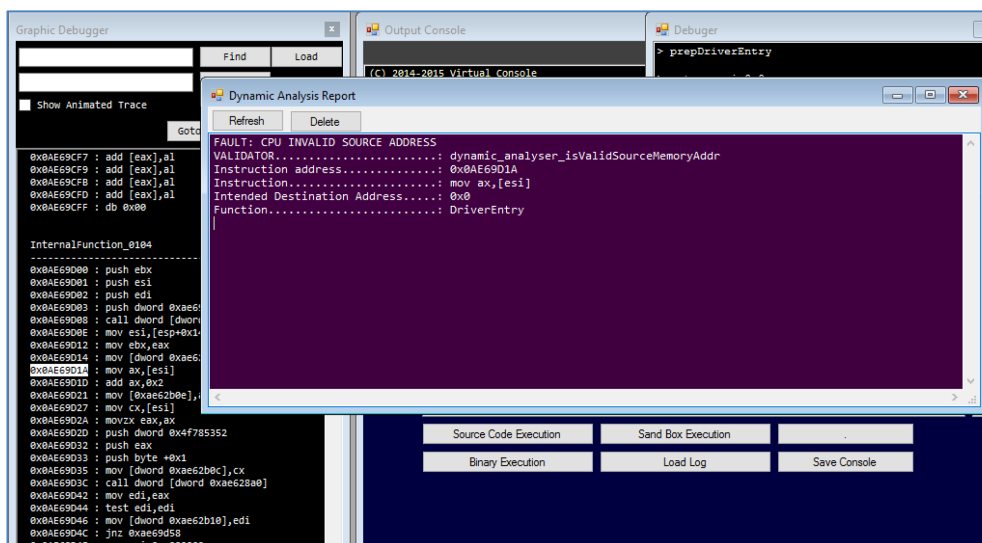


Figure AnII-6: Dynamic Report

BIBLIOGRAPHY

[1] B. Murphy, “Automating software failure reporting”, ACM Queue, Vol. 2, N.8, 2004.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating system errors”, In Proceedings of the Symposium on Operating Systems Principles, pp. 73-88, October 2001.

[3] L. Reveillere and G. Muller, “Improving Driver Robustness: an Evaluation of the Devil Approach”, In Proceedings of the International Conference on Dependable Systems and Networks, pp. 131-140, July 2001.

[4] “Net market share”, http://www.netmarketshare.com, accessed December 2016.

[5] D. Simpson, “Windows XP Embedded with Service Pack 1 Reliability”, Technical Report, Microsoft Corporation, January 2003.

[6] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing”, IEEE Transactions on Dependable and Secure Computing, January-March 2004.

[7] “ATIS Telecom Glossary 2007”, “Fault definition”, http://www.atis.org/glossary/definition.aspx?id=7926, accessed December 2016

[8] “Federal Standard 1037C”, “Fault definition”, http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm, accessed December 2016.

[9] “IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance, Dependability definition”, http://www.dependability.org/wg10.4/, accessed September 2016.

[10] R. Iyer and D. Rossetti, “A measurement-based model for workload dependence of CPU errors”, IEEE Transactions on Computers, vol. C-35, pp. 511-519, June 1986.

[11] E. Czeck and D. Siewiorek, “Effects of transient gate-level faults on program behaviour”, In Proceedings of the 20th Symposium on Fault-Tolerant Computing, pp. 236-243, June 1990.

[12] K. Goswami and R. Iyer, “A simulation-based study of a triple modular redundant system using DEPEND”, In Proceedings of the 5th International Conference on Fault- Tolerant Computing Systems, pp. 300-311, September 1991.

[13] E. Jenn, J. Arlat, M. Rimtn, J. Ohlsson, and J. Karlsson, “Fault Injection into VHDL Models: The MEFISTO Tool”, In Proceedings 24th International Symposium on Fault-Tolerant Computing, pp. 66-75, June 1994.

[14] V. Sieh, 0. Tschache, and F. Balbach, “VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions”, In Proceedings of the 27th International Symposium on Fault-Tolerant Computing, pp. 32 -36, June 1997.

[15] C. Kwang-Ting, H. Shi-Yu, and D. Wei-Jin, “Fault Emulation: A New Methodology for Fault Grading”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, N.10, pp. 1487-1495, 1999.

[16] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, “Fault Injection for Dependability Validation - A Methodology and Some Applications”, IEEE Transactions on Software Engineering, 16 (2), pp. 166-182, February 1990.

[17] M. Hsueh, T. Tsai, and R. Iyer, “Fault Injection Techniques and Tools”, IEEE Computer Vol. 30, pp. 75-82, April 1997.

[18] P. Folkesson, S. Svensson, and J. Karlsson, “A Comparison of Simulation Based and Scan Chain Implemented Fault Injection”, In Proceedings. 28th International Symposium on Fault-Tolerant Computing, pp. 284-293, June 1998.

[19] P. Folkesson, “Assessment and Comparison of Physical Fault Injection Techniques”, Ph.D. thesis, Chalmers University of Technology, 1999.

[20] H. Madeira, M. Rela, and J. G. Silva, “RIFLE: A General Purpose Pin-Level Fault Injector”, In Proceedings of the European Dependable Computing Conference, Springer LNCS, Vol. 852, pp.199-216, 1994.

[21] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. Leber, “Comparison of Physical and Software-Implemented Fault Injection Techniques”, IEEE Transactions on Computers, Vol.52, N.9, pp.1115-1133, September 2003.

[22] A. Rajabzadeh, S. G. Miremadi, and M. Mohandespour, “Experimental Evaluation of Master/Checker Architecture Using Power Supply and Software-Based Fault Injection”, in Proceedings of the 10th IEEE Int. On-Line Testing Symposium, pp. 239-44, July 2004.

[23] IEEE-ISTO 5001, “The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface V2.0”, IEEE-Industry Standards and Technology Organization, Piscataway, NJ 08854 USA, December 2003.

[24] IEEE Standard 1149.1-2001 “IEEE Standard Test Access Port and Boundary-Scan Architecture”, IEEE, Piscataway, NJ 08854 USA, 2001.

[25] NXP, “Background debug mode”, http://www.nxp.com/search?output=xml_no_dtd&proxystylesheet=nxp_search_style_fe&filter=0&getfields=*&rc=1&sort=date%3AD%3AL%3Ad1&oe=UTF-8&ie=UTF-8&ud=1&lang_cd=&wc=200&wc_mc=1&exclude_apps=1&site=nxp_en&dnavs=inmeta%3AAsset_Type%3DDocuments&client=nxp_search_documents&q=Background+debug+mode+inmeta%3AAsset_Type%3DDocuments, accessed December 2016.

[26] U. Gunneflo, J. Karlsson, and J. Torin, “Evaluation of error detection schemes using fault injection by heavy-ion radiation”, In Proceedings of the International Fault Tolerance Computer Symposium, FTCS-19, pp.340-347, June 1989.

[27] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin, “FIAT - Fault Injection Based Automated Testing Environment”, In Proceedings 18th International Symposium On Fault-Tolerant Computing, pp. 102-107, 1988.

[28] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. “Ferrari: A tool for the Validation of System Dependability Properties”, In Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 336-344, June 1992.

[29] W. I. Kao, R. K. Iyer, and D. Tang, “FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System behaviour under Faults”, IEEE Transactions on Software Engineering, pp. 1105-1118, Vol 19. No. 11, November 1993.

[30] W. I. Kao and R. K. Iyer, “DEFINE: A Distributed Fault Injection and Monitoring Environment”, Workshop on Fault-Tolerant Parallel and Distributed Systems, June 1994.

[31] S. Han, H.Rosenberg, and K.Shin, “DOCTOR: an Integrated Software Fault Injection Environment for Distributed Real-Time Systems”, In Proceedings of the Computer Performance and Dependability Symposium, pp. 204-213, 1995.

[32] T. Tsai and R. Iyer, “Measuring Fault Tolerance with the FTAPE Fault Injection Tool”, In Proceedings of the 8th International Conference Modelling Techniques and Tools for Computer Performance Evaluation, pp. 26-40, 1995.

[33] J. Carreira, H. Madeira, and J. G. Silva, “Xception: Software Fault Injection and Monitoring” in Processor Functional Units, IEEE Transactions on Software Engineering, Vol. 24, No. 2, February 1998.

[34] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, “GOOFI: Generic Object-oriented Fault Injection Tool”, In Proceedings of the International Conference on Dependable Systems and Networks, pp. 83-88, July 2001.

[35] L. T. Young, R. K. Iyer, K. K. Goswami, and C. Alonso, “A Hybrid Monitor Assisted Fault Injection Environment”, IFIP Working Conference on Dependable Computing for Critical Applications, Vol. 8, pp. 281-302, September 1992.

[36] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, “Comparing Operating Systems Using Robustness Benchmarks”, In Proceedings of the Symposium on Reliable and Distributed Systems, pp. 72-79, October 1997.

[37] “IEEE Standard Glossary of Software Engineering Terminology”, http://standards.ieee.org/findstds/standard/610.12-1990.html, accessed December 2016.

[38] A. Kalakech, T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun, “Benchmarking Operating System Dependability: Windows 2000 as a Case Study”, In Proceedings of the Pacific Rim International Symposium of Dependable Computing, pp.261-270, March 2004.

[39] M. Vieira and H. Madeira, “A Dependability Benchmark for OLTP Application Environments”, In Proceedings of the 29th International Conference on Very Large Databases, Vol. 29, pp. 742-753, September 2003.

[40] P. Oehlert, “Violating Assumptions with Fuzzing”, IEEE Security & Privacy, pp. 58-62, March/April 2005.

[41] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, “Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”, Computer Science Technical Report 1268, Univ. of Wisconsin-Madison, May 1998.

[42] G. Carrette, “CRASHME: Random input testing” (no publication available) http://people.delphi.com/gjc/crashme.html, accessed July 2013.

[43] J. DeVale, P. Koopman, and D. Guttendorf, “The Ballista Software Robustness Testing Service”, In Proceedings of the 16th International Conference of Testing Computer Software, pp. 33-42, 1999.

[44] M. Rodríguez, F. Salles, J. Fabre, and J. Arlat, “MAFALDA: Microkernel Assessment by Fault Injection and Design Aid”, In Proceedings of the European Dependable Computing Conference, Springer LNCS, Vol. 1667, pp. 143-160, 1999.

[45] P. Koopman and J. DeVale, “Comparing the Robustness of POSIX Operating Systems”, In Proceedings of the 29th International Symposium on Fault-Tolerant Computing, pp. 30-37, June 1999.

[46] C. Shelton, P. Koopman, and K. D. Vale, “Robustness Testing of the Microsoft Win32 API”, In Proceedings of the International Conference on Dependable Systems and Networks, pp. 261-270, June 2000.

[47] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities”, Communications of the ACM, 33(12):32–44, 1990.

[48] T. Biege, “Radius Fuzzer”, http://www.beyondsecurity.com /dynamic_fuzzing_testing_remote_authentication_dial_in_user_service_RADIUS_protocol, accessed December 2016.

[49] A. Greene, “SPIKEfile”, http://packetstormsecurity.com/files/39625/SPIKEfile.tgz.html, accessed December 2016.

[50] M. Sutton, “FileFuzz”, https://packetstormsecurity.com/files/39626/FileFuzz.zip.html, accessed December 2016.

[51] A. Ghosh, M. Schmid, and V. Shah, “Testing the robustness of Windows NT software”, In Proceedings of the 9th International Symposium on Software Reliability Engineering, pp. 231-235, November 1998.

[52] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles, “Dependability of COTS Microkernel-Based Systems”, IEEE Transactions on Computers, Vol. 51, No 2, pp. 138-163, 2002.

[53] A. Albinet, J. Arlat, and J.-C. Fabre, “Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel”, In Proceedings of the International Conference on Dependable Systems and Networks, pp.867-876, June 2004.

[54] J. Durães and H. Madeira, “Characterization of Operating Systems Behaviour in the Presence of Faulty Drivers through Software Fault Emulation”, In Proceedings of the Pacific Rim International Symposium of Dependable Computing, pp. 201-209, December 2002.

[55] A. Johansson and N. Suri, “Error Propagation Profiling of Operating Systems”, In Proceedings of the International Conference on Dependable Systems and Networks, pp. 86-95, July 2005.

[56] “PREfast for Drivers”, https://msdn.microsoft.com/windows/hardware/drivers/devtest/code-analysis-for-drivers, accessed December 2016.

[57] “Static Driver Verifier”, https://msdn.microsoft.com/windows/hardware/drivers/devtest/static-driver-verifier, accessed December 2016.

[58] T. Ball and S. Rajamani, “The SLAM project: debugging system software via static analysis”, In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 1-3, January 2002.

[59] “Device Path Exerciser”, https://msdn.microsoft.com/en-us/library/ff544856.aspx, accessed December 2016.

[60] “IoSpy and IoAttack”, https://msdn.microsoft.com/windows/hardware/drivers/devtest/iospy-and-ioattack, accessed December 2016.

[61] “Plug and Play Driver Test”, https://msdn.microsoft.com/en-us/library/ff550385.aspx, accessed December 2016.

[62] Microsoft Corporation, “Microsoft Portable Executable and Common Object File Format Specification”, February 2005.

[63] “Fuzzers”, http://malsecure.blogspot.pt/2006/11/fuzzers-ultimate-list.html, accessed December 2016.

[64] “Airpcap”, https://support.riverbed.com/content/support/software/steelcentral-npm/airpcap.html, accessed December 2016.

[65] “Madwifi driver”, http://madwifi-project.org/, accessed December 2016.

[66] “Lorcon project”, https://github.com/lunixbochs/lorcon, accessed December 2016.

[67] E. Marsden, J. C. Fabre, and J. Arlat, “Dependability of CORBA Systems: Service Characterization by Fault Injection”, In Proceedings of the 21st International Symposium on Reliable Distributed Systems, pp. 276-285, June 2002.

[68] J. Pan, P. Koopman, D. Siewiorek, Y. Huang, R. Gruber, and M. L. Jiang, “Robustness Testing and Hardening of CORBA ORB Implementations”, In Proceedings of the International Conference on Dependable Systems and Networks, pp. 141-150, June 2001.

[69] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, “Construction of a Highly Dependable Operating System”, In Proceedings of the 6th European Dependable Computing Conference, pp. 18-20, October 2006.

[70] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, "Reorganizing UNIX for Reliability". In Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference, pp. 81-94, September 2006.

[71] V. Ganapathy, M. Renzelmann, A. Balakrish-nan, M. Swift, and S. Jha, "The design and implementation of micro drivers", In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 168-178, March 2008.

[72] S. Butt, V. Ganapathy, M. M. Swift, and C. Chang, "Protecting Commodity Operating System Kernels from Vulnerable Device Drivers", In Proceedings of the 25th Annual Computer Security Applications Conference, pp. 301-310, December, 2009.

[73] M. Renzelmann and M. Swift, "Decaf: Moving Device Drivers to a modern language", In Proceedings of the 2009 USENIX Annual Technical Conference, 2009.

[74] J. Santos, Y. Turner, G.(John) Janakiraman, and Ian Pratt, "Bridging the gap between hardware and software techniques for i/o virtualization", In Proceedings of USENIX Annual Technical Conference, 2008.

[75] A. Menon, S. Schubert, and W. Zwaenepoel. "Twin Drivers: semiautomatic derivation off a stand safe hypervisor network drivers from guest OS drivers", In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, 2009.

[76] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems", In Proceedings of the 19th ACM Symposium on Operating Systems Principles, October 2003.

[77] M. Castro, M. Costa, J. P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation", In Proceedings of the 22nd ACM Symposium on Operating Systems Principles, October 2009.

[78] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation", In Proceedings of the 14th Symposium on Operating Systems Principles, pp. 203-216, December 1993.

[79] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Fault isolation for Device Drivers", In Proceedings of International Conference on Dependable Systems and Networks, 2009.

[80] A. Kadav, M. Lmann, and M.Swift, "Tolerating Hardware Device Failures in Software", In Proceedings of the 22nd Symposium on Operating systems principles, pp. 59-72, 2009.

[81] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing Closed-Source Binary Device Drivers with DDT", USENIX Annual Technical Conference, June 22-25, 2010.

[82] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, "Dealing with Disaster: Surviving Misbehaved Kernel Extensions", In Proceedings of 2nd Operating Systems Design and Implementation, 1996.

[83] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces", In Proceedings of the 7th Operating Systems Design and Implementation, 2006.

[84] L. Ryzhyk, P. Chubb, IhorKuz, and G. Heiser, “Dingo: Taming Device Drivers”. In Proceedings of the 4th EuroSys Conference, April 2009.

[85] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, “SafeDrive: Safe and recoverable extensions using Language-Based techniques”, In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, pp. 45-60, November 2006.

[86] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Software fault isolation with API integrity and multi-principal modules”, In Proceedings of the 23rd ACM Symposium on Operating Systems Principles pp. 115-128, 2011.

[87] L. Ryzhyk, P. Chubb, I. Kuz, E. Sueur, and G. Heiser, “Automatic Device Driver synthesis with Termite”, in Proceedings of the 22nd ACM Symposium on Operating Systems Principles, October 2009.

[88] Lea Wittie, “Laddie: Language for Automated Device Drivers”, Bucknell TR#08-2 Tech Report 2008.

[89] L. Ryzhyk, Y. Zhu, and Heiser, GA, “The case for Active Device Drivers”, in First ACM Asia-Pacific Workshop on Systems, August 2010.

[90] G. Hunt and D. Brubacher, “Detours: Binary Interception of Win32 Functions”, In Proceedings of the Conference of USENIX Windows NT Symposium, 1999.

[91] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach, “Dynamic Program Analysis of Microsoft Windows Applications”, In Proceedings of the International Symposium on Performance Analysis of Systems & Software, 2010.

[92] M. Mendonça and N. Neves, “Robustness Testing of the Windows DDK”, In Proceedings of the International Conference on Dependable Systems and Networks, June 2007.

[93] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis”, In Proceedings of the International Conference on Information Systems Security, December 2008.

[94] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, “Static detection of vulnerabilities in x86 executables”, In Proceedings of Annual Computer Security Applications Conference, 2006.

[95] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, “Combining unit-level symbolic execution and system-level concrete execution for testing NASA software”, In Proceedings of the International Symposium on Software Testing and Analysis, July 2008.

[96] “The LLVM Compiler Infrastructure”, http://llvm.org, accessed December 2016.

[97] V. Chipounov and G. Candea, “Enabling Sophisticated Analysis of x86 Binaries with RevGen”, In Proceedings of the International Conference on Dependable Systems and Networks, June 2011.

[98] “Libpcap file format”, https://wiki.wireshark.org/Development/LibpcapFileFormat, accessed December 2016.

[99] “WireShark”, http://www.wireshark.org, accessed December 2016.

[100] “Sweex”, http://www.sweex.com/, accessed December 2016.

[101] "WinDbg", https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit, accessed December 2016.

[102] J. Passing, A. Schmitdt, M. Lowis, and A. Polze, "NTrace: Function Boundary Tracing for Windows on IA-32", In Proceedings of the Working Conference on Reverse Engineering, October 2009.

[103] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization". In Proceedings of the International Symposium on Code Generation and Optimization, March 2003.

[104] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development". In Proceedings of Summer USENIX. July, 1986.

[105] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an application program". In USENIX 1990 Summer Conference, pp. 87-95, June 1990.

[106] M. Rozier, A. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhause, "CHORUS distributed operating system". Computing Systems, Vol. 1, N.4, pp. 305-370, 1988.

[107] S. R, Schach, B. Jin, D. R. Wright, G. Heller, and A.Offutt, "Maintainability of the Linux Kernel", In IEE Proceedings – Software, Vol. 149, Issue 1, pp. 18-23, February 2002.

[108] G. Candea and A. Fox, "Recursive Restartability: Turning the Reboot Sledgehammer into a Scapel", In Proceedings of the 8th Workshop on Hot Topics in Operating Systems, May 2001.

[109] L. Seawright and R. MacKinnon, "VM/370 – A study of multiplicity and usefulness", IBM Systems Journal, Vol. 18, N.1, pp. 4-17, 1979.

[110] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization", In Proceedings of the 9th ACM Symposium on Operating Systems Principles, pp. 164-177, 2003.

[111] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization", In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 2-13, October, 2006.

[112] C. Waldspurger, "Memory Resource Management in VMWare ESX Server", In Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Vol. 36, pp. 181-194, 2002.

[113] R. Goldberg, "Survey of Virtual Machine Research", IEEE Computer Society Press, pp. 34-45, 1975.

[114] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the XEN virtual machine monitor", in Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand on IT Infrastructure, October, 2004.

[115] M. Seltzer, Y. Endo, C. Small, and K. Smith, "An introduction to the architecture of the VINO kernel", Harvard University Computer Science Technical Report 34-94, 1994.

[116] G. Necula, "Proof-carrying code", In Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106-119, January, 1997.

[117] J. Corbet, A. Rubini, and G. Kroah-Hartman, “Linux Device Drivers”, 3rd edition, O’Reilly Media, 2005.

[118] J. Cooperstein, “Writing Linux Device Drivers – a guide with exercises”, Jerry Cooperstein, 2009.

[119] A. Robbins, “Essential Linux Device Drivers”, Sreekrishnan Venkateswaran, Prentice Hall, 2008.

[120] “Clay Programming Language”, http://claylabs.com/clay, accessed December 2013.

[121] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, “Thorough Static Analysis of Device Drivers”, Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems pp. 73-85, Vol. 40, issue 4, October 2006.

[122] R. Bryant, “Graph-Based algorithms for Boolean Function Manipulation”, IEEE Transactions on Computers C-35(8), pp 677-691, 1986.

[123] M. Christodorescu and S. Jha, “Static Analysis of Executables to Detect Malicious Patterns”, In Proceedings of the 12th USENIX Security Symposium, 2003.

[124] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, “Semantics-Aware Malware Detection”, In Proceedings of the 2005 IEEE Symposium on Security and Privacy, pp. 32-46, 2005.

[125] J. Newsome, B. Karp, and D. Song, “Polygraph: Automatically Generating Signatures for Polymorphic Worms”, In Proc. of the 2005 IEEE Symposium. on Security and Privacy, pp. 226-241, 2005.

[126] C. Kruegel, W. Robertson, and G. Vigna, “Detecting Kernel-Level Rootkits Through Binary Analysis”, In Proceedings of the Annual Computer Security Applications Conf. (ACSAC), pp 91–100, December 2004.

[127] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, “Behavior-based spyware detection”. In Proceedings of the 15th USENIX Security Symposium, 2006.

[128] “Java Path Finder”, http://babelfish.arc.nasa.gov/trac/jpf, accessed December 2016.

[129] M. Swift, M. Annamalau, B. N. Bershad, and H. M. Levy, "Recovering Device Drivers", in ACM Transactions on Computer Systems, 24 (4), November 2006.

[130] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, “CIL: Intermediate language and tools for analysis and transformation of C programs”, In Proceedings of the 11th International Conference on Compiler Construction, 2002.

[131] “CWE list”, http://cwe.mitre.org/data/index.html, accessed December 2016.

[132] K. Tsipenyuk, B. Chess, and G. McGraw, “Seven pernicious kingdoms: A taxonomy of software security errors.”, IEEE Security & Privacy, Vol.3, pp. 81-84, December 2005.

[133] D. Engler, B. Chelf, A. Chou, and S. Hallem, “Checking system rules using system-specific programmer-written compiler extensions”, In Proceedings of the 4th Symposium on Operating Systems Design and Implementation, pp. 23-25, October 2000.

[134] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code", In Proceedings of the 18th ACM Symposium on Operating Systems Principles, pp. 57-72, October 2001.

[135] C. Artho and A. Biere, "Applying static analysis to large-scale, multi-threaded Java programs", In Proceedings of the Australian Software Engineering Conference, pp.68-75, August 2001.

[136] C. Artho and K. Havelund, "Applying Jlint to space exploration software", In Verification Model Checking and Abstract Interpretation, Vol 2937/2003 of Lecture Notes in Computer Science, pp. 297-308, Springer Berling/Heidelberg, 2004.

[137] D. Evans and D. Larochelle, "Improving security using "Extensible light weight static analysis", IEEE Software, pp. 42-51, February 2002.

[138] Y. Xie, A. Chou, and D. Engler, "Archer: using symbolic, path-sensitive analysis to detect memory access errors", In ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACMSIGSOFT International Symposium on Foundations of Software Engineering, pp.327-336, 2003.

[139] D. Hovemeyer and W. Pugh, "Finding bugs is easy", In Companion to the 19th Annual ACMSIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications, pp.92-106, October 2004.

[140] "Gramma Tech Code Sonar", https://www.grammatech.com/products/codesonar, accessed December 2016.

[141] W. R. Bush, J. D. Pincus, and D. J. Siela, "A static analyser for finding dynamic programming errors", Software Practice & Experience, Vol. 30, pp.775-802, May 2000.

[142] M. Das, S. Lerner and M. Seigle, "ESP: Path-sensitive program verification in polynomial time", In Proceedings of the Conference on Programming Language Design and Implementation, pp. 57-68, June 2002.

[143] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg and F. Rauch, "Goanna – a static model checker", In Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems, N. 4346 in Lecture Notes in Computer Science, August 2006.

[144] "Veracode", http://www.veracode.com/, accessed December 2016.

[145] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces", In Proceedings of the Workshop on Model Checking of Software, LNCS2057, pp. 103-122, May 2001.

[146] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world", Communications of the ACM, Vol. 53, N.2, 2010.

[147] C. Cadar, D. Dunbar, and D. R. Engler. "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs", In Proceedings of the Symposium on Operating Systems Design and Implementation, 2008.

[148] O. Frank and D. Strauss, "Markov Graphs", Journal of the American Statistical Association, Vol. 81, N.395, pp.832-842, 1986.

[149] R. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies", IEEE Transactions on Device and Materials Reliability, Vol. 5, N.3, pp.305-316, September 2005.

[150] M. Leitner, D. Wutte, J. Brandstotter, F. Aumayr, and HP. Winter, "Single-stage 5GHz ECR-multicharged ion source with high magnetic mirror ratio and blased disk", Review of Scientific Instuments, Vol.65, N.4, pp. 1091-1993, April 1994.

[151] M. Zenha-Rela and J. Cunha, "Exploiting the IEEE 1149.1 standard for software reliability evaluation in space applications", European Safety and Reliability Conference, pp.1459-1464, September 2006.

[152] P. M. Folkesson, "Assessement and Comparison of Physical Fault Injection, Techniques", Phd thesis, Chalmers University of Technology, 1999.

[153] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J.Reisinger, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", in Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications, pp.150-151, 1995.

[154] A. Brown, J. Traupman, P. Broadwell, and D. Patterson, "Practical Issues in Dependability Benchmarking", Proceedings of the second Workshop on Evaluating and Architecting System Dependability, 2002.

[155] H. Madeira and P. Koopman, "Dependability Benchmarking: making choices in an n-dimensional problem space", Proceedings of the first Workshop on Evaluating and Architecting System Dependability, 2001.

[156] "NASM: The netwide assembler", http://www.nasm.us, accessed December 2016.

[157] "Getting Started with Windows drivers", https://msdn.microsoft.com/en-us/library/windows/hardware/ff554690%28v=vs.85%29.aspx, accessed December 2016.

[158] "Programming Guide", https://msdn.microsoft.com/en-us/library/windows/hardware/hh406596%28v=vs.85%29.aspx, accessed December 2016.

[159] "Windows Programming Device Drivers Introduction", https://en.wikibooks.org/wiki/Windows_Programming/Device_Driver_Introduction, accessed December 2016.

[160] R. D. Reeves, "Windows 7 Device Drivers", Addison-Wesley Microsoft Technology Series, 2011.

[161] W. Oney, "Programming the Microsoft Windows Driver Model", 2nd Edition, Microsoft Press, 2003.

[162] P. Orwick, G. Smith, "Developing Drivers with Windows Driver Foundation", Microsoft Press, 2007.

[163] "Download kits for Windows hardware development", https://developer.microsoft.com/en-us/windows/hardware/download-kits-windows-hardware-development, accessed December 2016.

[164] "Windows Driver Samples", https://msdn.microsoft.com/windows/hardware/drivers/samples/index, accessed December 2016.

[165] "Windows Driver Code Samples", https://msdn.microsoft.com/en-us/windows/hardware/dn433227.aspx, accessed December 2016.

[166] "Windows Driver Kit (WDK) 8.0 Samples", https://code.msdn.microsoft.com/windowshardware/windows-driver-kit-wdk-80-e3161626, December 2016.

[167] "Bochs", http://bochs.sourceforge.net, accessed December 2016.

[168] "QEMU", https://github.com/qemu/qemu, accessed December 2016.

[169] "CLOC", https://github.com/AlDanial/cloc, accessed December 2016.

[170] "Intel 64 and IA-32 Architectures Software Developer's Manual, Vol 2 (2A,2B & 2C): Instruction Set Reference, A-Z", http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, accessed December 2016.

[171] "Instructions per second", https://en.wikipedia.org/wiki/Instructions_per_second, accessed December 2016.