A HYBRID APPROACH TO OPERATING SYSTEM DISCOVERY BASED ON DIAGNOSIS THEORY

by
François Gagnon

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario
June, 2010

Table of Contents

List of Tables **ix**

List of Figures **x**

List of Acronyms **xii**

Abstract **xiii**

Acknowledgements **xiv**

Chapter 1 Introduction **1**

1.1 Thesis Structure 2

1.2 Contributions 2

Chapter 2 State of the Art - Operating System Discovery **4**

2.1 What is Operating System Discovery? 4

2.1.1 Banner Grabbing 5

2.1.2 MIB & SNMP Agent 6

2.1.3 Behavioral Analysis 6

2.2 OSD Through Behavioral Analysis 6

2.2.1 OSD Tests 9

2.2.1.1 Type 11

2.2.1.2 Nature 12

2.2.1.3 Execution Mode 12

2.3 Passive Approach 13

2.3.1 Passive Tools 13

2.3.1.1 `p0f` 14

2.3.1.2 Other Passive Tools 17

2.3.2 Advantages and Limitations 17

2.3.2.1 Information Unavailability . . . 17
2.3.2.2 Restriction to Single-Packet Rules . . . 17
2.3.2.3 Lack of Memory . . . 18
2.4 Active Approach . . . 18
2.4.1 Active Tools . . . 19
2.4.1.1 Nmap . . . 20
2.4.1.2 Xprobe . . . 22
2.4.2 Advantages and Limitations . . . 24
2.4.2.1 Amount of Traffic Generated . . . 24
2.4.2.2 Malformed Traffic . . . 25
2.5 Obfuscating OS Discovery . . . 25
2.6 Discussion . . . 26

Chapter 3 Motivation 28
3.1 Why is OSD important? . . . 28
3.1.1 Single OS Query . . . 28
3.1.2 Group OS Query . . . 29
3.1.3 Exact OS Query . . . 29
3.2 Evaluating Current OSD Tools . . . 30
3.2.1 Dataset . . . 30
3.2.2 OSD Tools . . . 31
3.2.3 Group OS Query . . . 31
3.2.3.1 Results . . . 33
3.2.4 Exact OS Query . . . 36
3.2.4.1 Results . . . 36
3.3 Discussion . . . 38

Chapter 4 Background - Diagnosis 40
4.1 What is Diagnosis? . . . 40
4.1.1 Diagnosis Problem Specification . . . 40
4.1.2 Diagnosis Terminology . . . 42

4.1.3 Four Diagnosis Families 44
4.1.4 Reiter's Model-Based Diagnosis 46
4.1.5 Rule-Based Diagnosis 49
4.1.5.1 Intuitive Meaning of the Rules 50
4.1.5.2 Handling Incomplete Knowledge 51
4.1.5.2.1 Unanticipated Explanatory Constituent . . 51
4.1.5.2.2 Unanticipated Causal Relation 52
4.1.5.2.3 Unanticipated Observation 52
4.1.6 Diagnosis properties 52
4.2 Candidate Generation 55
4.3 Candidate Elimination 57
4.3.1 Test Representation 57
4.3.1.1 Reiter's Test Representation 58
4.3.1.2 McIlraith's Test Representation 59
4.3.1.3 Outcome-Based Test Representation 59
4.3.2 Current Test Selection Strategy 62
4.4 Limitations 63

Chapter 5 Problem Statement 64
5.1 Objectives 64
5.1.1 A Better Tool 64
5.1.2 A Strong Theoretical Background 65
5.1.3 Systematic Collection of OS Fingerprints 65
5.2 Relevance 66
5.3 Methodology 66
5.4 Evaluation and Validation 67
5.4.1 Ideal OSD Evaluation 67
5.4.2 The Evaluation Dataset 68

Chapter 6 Towards Better Operating System Discovery 70
6.1 A Hybrid Approach to Operating System Discovery 70

6.1.1 The General Picture of Hybrid OS Discovery 71
6.1.2 The Passive Module 72
6.1.2.1 Querying the Knowledge Base 73
6.1.3 The Active Module 73
6.1.4 Hybrid OSD as a Diagnosis Task 74
6.1.4.1 Explanatory Constituents (CONST) 75
6.1.4.2 Observations (OBS) 75
6.1.4.3 System Description (SD) 75
6.1.4.4 Tests (TEST) 76
6.1.4.5 Properties of the OSD Diagnosis Task 77
6.1.4.6 Candidate Generation Algorithm for OSD 78
6.1.4.6.1 Algorithm Analysis Parameters 79
6.1.4.6.2 Conflict Sets Generation 80
6.1.4.6.3 Hitting Sets Generation 81
6.1.4.6.4 Impact 83
6.2 Extending the Theory of Diagnosis 83
6.2.1 Query-Based Extension 84
6.2.1.1 Diagnosis Queries 85
6.2.1.2 Meaningfulness of Diagnosis Queries 86
6.2.1.2.1 Medical Diagnosis 86
6.2.1.2.2 Engineering Domain 86
6.2.1.2.3 Discussion 87
6.2.1.3 Usefulness of Diagnosis Queries 87
6.2.2 Test Selection Strategies 89
6.2.2.1 Properties 89
6.2.2.2 Single Candidate Query 90
6.2.2.2.1 Assumptions 90
6.2.2.2.2 Solution Structure 92
6.2.2.2.3 Optimal Characterizability 93
6.2.2.2.4 Solvability 93

6.2.2.2.5 Solution Verifiability 93
6.2.2.2.6 Complexity 93
6.2.2.2.7 Approximability 95
6.2.2.3 Exact Candidate Query 101
6.2.2.3.1 Solution Structure 101
6.2.2.3.2 Optimal Characterizability 102
6.2.2.3.3 Solvability 102
6.2.2.3.4 Solution Verifiability 102
6.2.2.3.5 Complexity 103
6.2.2.3.6 Approximability 103
6.2.2.4 Summary 103
6.3 Virtual Network Experiment Controller 104
6.3.1 The General Problem 104
6.3.2 VNEC Architecture 106
6.3.2.1 Network Specification 106
6.3.2.2 Task Workflow Specification 107
6.3.2.3 Experiment Execution 109
6.3.3 OS Fingerprinting with VNEC 111
6.3.3.1 Reactive Events 111
6.3.3.2 Spontaneous Events 112
6.3.3.3 Limitations 113
6.4 Discussion 114

Chapter 7 Implementation & Evaluation 115

7.1 Implementation 115
7.1.1 OS Fingerprints Database 115
7.1.2 Passive Module 116
7.1.2.1 Using Prolog 116
7.1.3 Active Module 117
7.1.4 Greedy - Exact Candidate Query 118

7.1.5 Greedy - Single Candidate Query . . . 118
7.1.6 Brute Force - Single Candidate Query . . . 118
7.2 Experiment Results for Test Selection Strategies . . . 119
7.2.1 Experiment Setup . . . 119
7.2.2 Results . . . 119
7.2.3 Summary . . . 121
7.3 Evaluation Experiment Results . . . 121
7.3.1 Exact Candidate Query . . . 122
7.3.1.1 Correctness . . . 122
7.3.1.2 Imprecision . . . 123
7.3.1.3 Traffic Generated . . . 124
7.3.2 Group Candidate Query . . . 124
7.3.2.1 Recall . . . 125
7.3.2.2 Precision . . . 126
7.3.2.3 Traffic Generated . . . 127
7.4 Discussion . . . 128
7.4.1 Deployment . . . 128

Chapter 8 Conclusion **130**
8.1 Thesis Summary . . . 130
8.2 Conclusion . . . 131
8.3 Contributions . . . 132
8.3.1 Tools . . . 133
8.4 Future Work . . . 133

Bibliography **136**

Appendix A OSD Tests **143**
A.1 Test Descriptions . . . 143
A.2 Test Results . . . 147

Appendix B Evaluation Dataset **156**

List of Tables

Table 2.1 OS Discovery Tests 9
Table 2.2 Advantages/Inconvenients of OSD Approaches/Tools 26

Table 4.1 SD for the Full Adder 48
Table 4.2 Diagnosis Candidates for Example 4.4 49
Table 4.3 Minimal Diagnosis Candidates for Example 4.4 49

Table 6.1 Advantages/Inconvenients of OSD Approaches/Tools 72
Table 6.2 Diagnosis Properties for OS Discovery 79
Table 6.3 Discriminant Power 100

Table 7.1 Comparing Test Selection Strategies 120
Table 7.2 Packet Injection Summary for Single Candidate Query 124
Table 7.3 Packet Injection Summary for Group Candidate Query 127

Table A.1 Fingerprints for the RstAck Tests 147
Table A.2 OS Fingerprint Associations for Test RstAck 148

Table B.1 Exploit List 156
Table B.2 Target List 160

List of Figures

Figure 2.1 Mail Server Banner 5
Figure 2.2 Kernel Architecture (source: Figure 2-1 in [78]) 7
Figure 2.3 p0f Signature File Format 15
Figure 2.4 Nmap Signature File Format 21
Figure 2.5 Xprobe Signature File Format 23

Figure 3.1 Automatic Evaluation Process - OSD Tools 32
Figure 3.2 ContextOS Algorithm 33
Figure 3.3 OSD Tools Recall 34
Figure 3.4 OSD Tools Precision 35
Figure 3.5 OSD Tools Correctness 37
Figure 3.6 OSD Tools Imprecision 38

Figure 4.1 Simple Diagnosis Tool 42
Figure 4.2 Full Adder Device 47
Figure 4.3 Naive Algorithm to Compute Diagnosis Candidates in Multiple Faults 55
Figure 4.4 Test Execution 61
(a) Single Test 61
(b) Test Sequence 61

Figure 6.1 General Candidates Generation Algorithm 80
Figure 6.2 Conflict Sets Generation Algorithm 81
Figure 6.3 Hitting Sets Generation Algorithm 82
Figure 6.4 Query-Based Diagnosis Tool Behavior 84
Figure 6.5 Test Execution for Single Candidate Query 91
(a) Single Test 91
(b) Test Sequence 91
(c) Simplified Test Sequence 91

Figure 6.6 Greedy Algorithm for Test Selection 99
Figure 6.7 Tree Comparison Metrics 102
Figure 6.8 Client-to-Server Attack Experiment 106
Figure 6.9 Snapshot of VNEC - Network Specification 107
Figure 6.10 Examples of Task Workflows 109
(a) Linear Task Workflow 109
(b) Non-Linear Task Workflow 109
Figure 6.11 VNEC Communication Architecture 110

Figure 7.1 Content of passiveOSFingerprinting.pl 117
Figure 7.2 Content of testPossibleOutcomes.txt 117
Figure 7.3 Correctness for the Exact Candidate Query 122
Figure 7.4 Imprecision for the Exact Candidate Query 123
Figure 7.5 Recall for the Group Candidate Query 125
Figure 7.6 Precision for the Group Candidate Query 126

List of Acronyms

OS: Operating System

OSD: Operating System Discovery

IDS: Intruction Detection System

TTL: Time To Live

DF: Don't Fragment

VM: Virtual Machine

NAT: Network Address Tranlation (or Translator)

BID: Bugtraq Identifier

SD: System Description

CONST: Explanatory Constituant

OBS: Observation

posd: (Tool name) Passive Operating System Discovery

aosd: (Tool name) Active Operating System Discovery

hosd: (Tool name) Hybrid Operating System Discovery

VNEC: (Tool name) Virtual Network Experiment Controller

Abstract

Motivated by the increasing importance of knowing which operating systems are running in a given network, we evaluated operating system discovery (OSD) tools. The results indicated a serious lack of accuracy in current OSD tools.

This thesis proposes a new OSD approach which addresses the limitations of existing approaches and leads to a more flexible, less intrusive, and much more accurate tool. Moreover, unlike existing OSD tools which are completely ad hoc, our approach is formal and follows the principles of diagnosis problem solving. This formalization allows us to characterize the complexity of OSD, use well-tested algorithms, and benefits from numerous extensions.

To fully address the needs of OSD, we generalize the theory of diagnosis with a query-based extension. This extension leads to a spectrum of test selection algorithms to solve each query.

Acknowledgements

First and foremost, I want to thank professor Esfandiari for being such a wonderful thesis supervisor. Thanks for giving me all those exciting opportunities. Thanks for allowing me to explore interesting areas that were not directly related, at least at first sight, to my topic. Thanks for providing the resources and putting all the extra efforts so my projects could progress rapidly, and thanks for letting me lead all those interesting projects. But, most importantly, thanks for letting me make my own mistakes without ever saying "I told you so". I really feel I have grown a lot during those four years, and this is mainly due to the environment you provided for me.

Part of the work in this thesis has been done in collaboration with the Network Security research group at the Communications Research Centre Canada (CRC). More precisely, the evaluation experiments of Chapters 3 and 7 were executed using their *vlab* environment. Frédéric Massicotte, from CRC, is responsible for making this fruitful collaboration possible. On a more personal note, thanks to both Frédéric and Mathieu for bringing me to Ottawa for my Ph.D.

Nothing would have been possible without the unconditional support of Cath. You made all of my hard decisions look so easy. I truly hope we'll have many other interesting adventures together. Thanks to all of our new friends here in Ottawa: from volley-ball, waterski, work, and school. Thanks to you for making us feel at home here, but most of all, thanks for making it so hard for us to leave!!!

The last words go to my parents and familly who helped me in so many ways. Thanks for believing in my dreams, thanks for showing me the way, thanks for all your advices. But most importantly, thanks for being such great role models.

To my dad;
for it all started with a chess game!

To Cath, my love, my soulmate;
for my home is wherever we are, together!

Computer Science is no more about computers than astronomy is about telescopes.
-Edsger Dijkstra

Chapter 1

Introduction

Since we cannot know all that there is to be known about anything, we ought to know a little about everything.
-Blaise Pascal

Operating system discovery (OSD) is the problem of finding which operating system is running on the computers of a given network by looking at the communication from/to those computers [80]. The importance of automatically gathering this kind of knowledge is growing rapidly as networks are getting larger, more heterogeneous and users have more control over their own workstation.

Unfortunately, current OSD tools are unreliable in the sense that they often provide the wrong information. This unreliability is particularly problematic when an automated process (e.g., filtering alarms from intrusion detection systems) relies on the output of an OSD tool.

Current OS discovery tools have several limitations, most being the result of poor engineering. In this thesis we propose a new approach to operating system discovery designed around a strong knowledge representation component. We have three main objectives regarding the development of our new operating system discovery tool:

- We want our tool to be more accurate than every other existing tool, especially for the task of intrusion detection context gathering.

- We want to study the computational complexity of OSD and take advantage of effective and well-tested algorithms. Both can be achieved by finding an appropriate and well-studied formalism to represent OSD.

- Current OSD tools rely on users submitting new fingerprints through a web site in an ad hoc manner. We want to provide a systematic (and automated) way of collecting OS fingerprints and incorporating them in our tool.

1.1 Thesis Structure

The thesis is structured as follows:

Chapter 2 introduces operating system discovery and presents the two classical approaches: *active* (where probes can be sent to the target to stimulate a reaction) and *passive* (where no packet can be injected). It also presents the state of the art of OSD tools. Of particular importance is the discussion about the advantages and limitations of current OSD tools/approaches. This discussion provides insight as to why current OSD tools are inaccurate and how we could build a better one.

Chapter 3 provides the motivation as to why operating system discovery is important. In this chapter, we study the impact of using contextual information for the reduction of non-critical alarms in intrusion detection systems. One possibility is to use information about the target's operating system to establish the likelihood of success of a specific attack. Among other things, Chapter 3 shows that existing OSD tools are not adequate for gathering IDS context.

Since the new OS discovery approach presented in the thesis is formalized using the theory of diagnosis, chapter 4 provides the necessary background regarding that theory.

Chapter 5 states the problem addressed through this thesis and discusses our principal objectives as well as our evaluation process.

Chapter 6 describes our three important contributions: an extension to the theory of diagnosis, our new hybrid approach to OS discovery, and our virtual environment to gather OS fingerprints.

Chapter 7 discusses an implementation of our approach. Moreover, it evaluates our tool, comparing it against current state of the art OS discovery tools.

Some interesting pointers towards future work are discussed in the closing chapter.

1.2 Contributions

The main contribution of this thesis is a *hybrid* approach to operating system discovery combining both passive and active techniques by using diagnosis as a knowledge representation framework [22]. We attack the OSD problem from both practical and

theoretical angles. First, we want to develop a tool that is better (more accurate, more flexible, and less intrusive) than the existing ones. But, we also want to have a good understanding of the underlying complexity of the problem and, when possible, to reuse or adapt existing algorithms. In our situation, logical diagnosis provides an adequate theory to represent OSD.

Another important contribution is an extension to the theory of diagnosis in order to obtain the flexibility required by our OSD tool [20]. This extension relies on a query-based approach to diagnosis and leads to the definition of a spectrum of test selection strategies. It is important to note that this flexibility requirement is not exclusive to our OSD application; many other diagnosis problems should indeed benefit from this extension.

Moreover, we have two other contributions:

- Since our primary motivation is security, we provide a study of the impact of contextual information on the reduction of non-critical alarms in intrusion detection systems [25]. In particular, we measure the impact of using information about the target configuration (mainly the operating system) to support IDS. We also establish that existing OSD tools are not good enough to gather that information.

- We also address the problem of gathering OS fingerprints by developing a tool to automatically execute network experiments in a virtual environment [16]. This eliminates the need of having hundreds of physical machines to fingerprint, each running a different OS. Our tool is general enough to be used for other purposes such as studying virus propagation patterns and target behavior while under attack.

Chapter 2

State of the Art - Operating System Discovery

Our students, twenty years from now, will be using the concepts, tools, and technology, that they will learn fifteen years from now.
-Nils Nilson

This chapter provides an introduction to operating system discovery (OSD) and presents the state of the art for that field. First, Section 2.1 explains what is OS discovery and briefly describes the different techniques. Section 2.2 explains in more detail the OSD technique used in this thesis, i.e., remote OSD through behavioral analysis of the host. Sections 2.3 and 2.4 present some existing OSD tools and discuss their limitations. Finally, Section 2.5 discusses some countermeasures to prevent OS discovery.

2.1 What is Operating System Discovery?

Remote operating system discovery (OSD) consists of inferring which operating system is running on a given host by analyzing its network communication patterns [80].

It would be possible to manually fetch the information (e.g. using commands like `uname -a` on Linux or looking at the graphical interface) and maintain a database containing the operating system (OS) of each host. Unfortunately, a manual approach is definitely not adequate for large, heterogeneous, and dynamic networks. Indeed, visiting hundreds of workstations to manually obtain their OSes is a tedious task. Moreover, with dynamic networks (e.g., devices joining and leaving continuously through wireless access points and users modifying their workstation by themselves) it is nearly impossible to manually maintain an up-to-date database containing the OS of each computer. For these reasons, we do not consider the manual approach any further in this thesis. We instead turn to automated ways of determining the

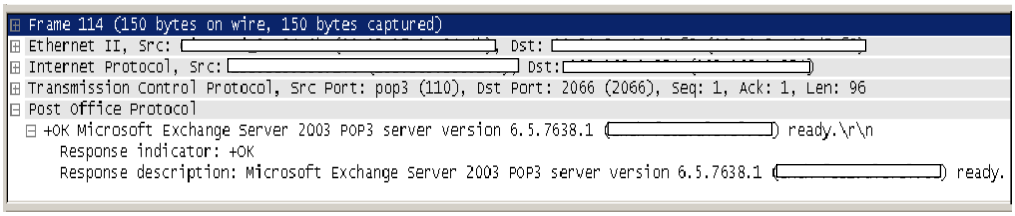


Figure 2.1: Mail Server Banner

operating system of a computer in a remote way.

Three different remote automated techniques are discussed in the literature: banner grabbing [57], using MIB & SNMP agents [53], and behavioral analysis [74]. Each technique is briefly described below. A more extensive presentation of the behavioral analysis technique, which is the one used in this thesis, is provided in Section 2.2.

2.1.1 Banner Grabbing

Some applications advertise, by default, the operating system of their host. This is done in plain text in the payload of the packets used to establish a connection. It is therefore possible to grab the advertisement banner and obtain the OS, see Example 2.1.

Example 2.1 (banner grabbing)
Figure 2.1 illustrates the banner provided by the pop mail server of the computer science department at Université Laval. It is clear from the banner that the mail server is Microsoft Exchange. From there we can conclude that the mail server is running a Windows operating system. ♢

Banners are easily modified on several client applications. For instance, the user-agent banner in Internet Explorer can be changed via an entry in the registry table and Firefox even provides a configuration tool to modify it (available at the URI `about:config`). Moreover, it is considered good practice to change the default configuration on server applications [1], either advertising the wrong OS or not advertising anything at all. This helps prevent leaking sensitive information regarding the server configuration. As a consequence, banner grabbing is considered to be unreliable [36]. For this reason, we do not consider banner grabbing any further in this thesis.

2.1.2 MIB & SNMP Agent

It is possible to query the SNMP agent of a network device to obtain its sysObjectID MIB entry [53]. This entry provides information about the device OS (or firmware). However, this technique requires the remote host to run an SNMP agent, which is not the default configuration of workstation systems. As a consequence, this is not a general solution to OS discovery.

2.1.3 Behavioral Analysis

Finally, it is possible to analyze the communication behavior of a computer to identify its operating system. In several situations, two different OSes will not exhibit the same behavior, thus providing a way to distinguish them.

This technique relies on communication patterns, i.e., network packets. Thus, it does not require any specific configuration on the computers. Moreover, since the behavior is encoded in the operating system (at the kernel level), behavioral analysis provides a fundamental way of identifying the OS. While it is still possible to tamper with the communication behavior of a computer (see Section 2.5) it is neither easy nor recommended.

Since we believe it to be the best technique available, we focus entirely on OS discovery through behavioral analysis in the remainder of this thesis.

2.2 OSD Through Behavioral Analysis

An operating system (OS) is a software layer between the hardware and the users/ applications. The main component of an OS is the kernel which is actually responsible for the device management (i.e., interaction with the hardware) among other things (e.g., process and memory management). A part of the kernel, the TCP/IP stack, is dedicated to the communication on the network (see TCP/IP Protocol drivers in Figure 2.2).

When an application needs to send network packets, it will rely on the operating system to do so. For instance, when Internet Explorer needs to display the page `http://www.nmai.ca/research-projects`, it will ask the kernel to build and

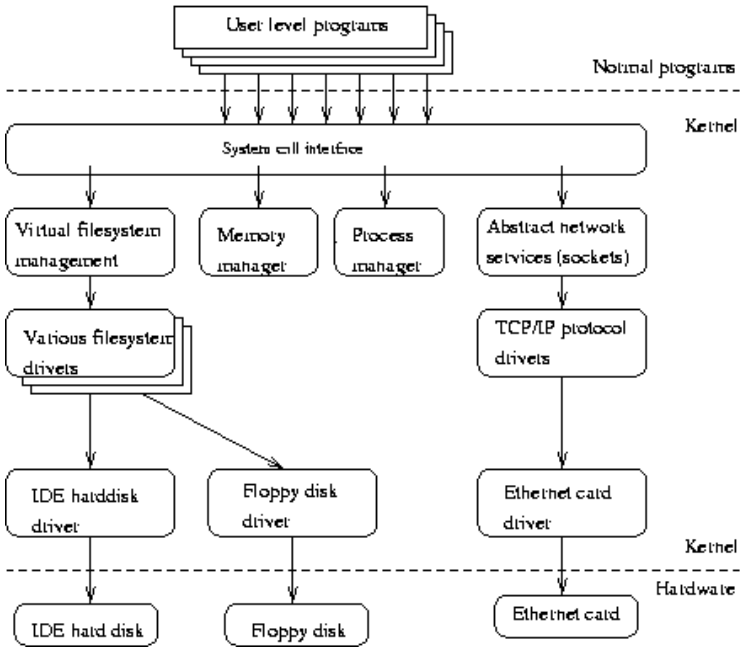


Figure 2.2: Kernel Architecture (source: Figure 2-1 in [78])

send the HTML request packet on its behalf by simply providing the destination (e.g., www.nmai.ca) and the content of the request (e.g., GET `/research-projects` HTTP/1.1). It is up to the TCP/IP stack to build the complete TCP packets, including the internet and link layer components.

The way a TCP/IP stack should build a packet is dictated by the communication protocol standards via Request For Comments (RFC) documents. There are two reasons why two unrelated TCP/IP stacks could behave differently:

- One has a bug and, as a result, does not respect protocol standards.
- A protocol is under-specified, and vendor-specific decisions to implement the protocol lead to different behaviors (see Example 2.2).

Example 2.2 (OS identification from TTL)
Every IP packet has a Time To Live (TTL) value which is decreased every time the packet traverses a router. RFC 791 *suggests* an initial TTL value of at least 40. As a result, not all OS vendors use the same initial TTL value. For instance, for the TTL in SYN packets: Linux uses 64, MacOS uses 255, most Windows versions use 128, but Windows NT 3.51 uses 32. ♢

Because different kernels (and even different versions of the same kernel) almost never have the exact same TCP/IP stack implementation, it is possible to associate an observed network behavior with a specific kernel. For instance, even though NetBSD 1.6 is based on the kernel of NetBSD 1.5, they do not behave identically when responding to the ICMP Echo Request.

It is possible that two consecutive kernel releases have the same TCP/IP stack (or TCP/IP stacks with identical behavior), in that case they would be undistinguishable remotely based on their network behavior (this seems to be the case with Windows 2000 Sp2 and Windows 2000 Sp2). However, there are three reasons why the TCP/IP stack of two consecutive kernel releases are often different:

- A bug with respect to the TCP/IP protocol specification was detected and fixed from one kernel version to the other.

- A security vulnerability was fixed between the two kernel versions. Since security vulnerabilities are usually related to network communication, fixing a vulnerability often leads to some modifications in the TCP/IP stack.

- Kernels need to adapt their TCP/IP stack to maintain compatibility with the fast evolving Internet [46]. For instance, the ICMP Info Request protocol is now considered obsolete, thus recent TCP/IP stacks do not consider these requests anymore (while older ones still respond). Another example would be the need for recent kernels to support IPv6 [45].

Each situation where two TCP/IP stacks do not behave identically provides a possibility to extract some information about the current OS. These opportunities are called OS discovery *tests* [2]. For each test, we can identify the behavior of a specific OS (more precisely its TCP/IP stack); we then talk about the *fingerprint* of that OS with respect to a given test [74]. Unlike human fingerprints, OS fingerprints are not unique: Windows 2000 and XP have the same fingerprints with respect to many tests. For that reason, it is important to consider several tests when trying to identify an operating system. Below, we present the tests considered in this thesis.

2.2.1 OSD Tests

Table 2.1 presents the 21 tests considered in this thesis. This is a subset of the tests presented in [2] and it represents the tests commonly used by existing OSD tools. More detail about these tests can be found in Appendix A. Each test has three properties: its type, the nature of the resulting traffic, and its execution mode. The properties are discussed below.

Table 2.1: OS Discovery Tests

Property / Test	Type	Nature	Execution Mode
Test-1 TCP Syn	Single-Packet	Standard	Passive
Test-2 ARP Request	Single-Packet	Standard	Both
Test-3 TCP ISN	Sample	Standard	Passive
Test-4 IP ID	Sample	Standard	Both
Test-5 TCP TS	Sample	Standard	Both
Test-6 ARP Retransmit	Sample	Standard	Both
Test-7 ICMP ID Seq	Sample	Standard	Passive
Test-8 SynAck	Stimulus-Response	Standard	Both
Test-9 RstAck	Stimulus-Response	Standard	Both
Test-10	Stimulus-	Standard	Both

Continued on next page

Table 2.1 – OS Discovery Tests

Property / Test	**Type**	**Nature**	**Execution Mode**
ICMP Unreach	Response		
Test-11 ICMP Echo	Stimulus-Response	Standard	Both
Test-12 ICMP Info	Stimulus-Response	Undetermined	Undetermined
Test-13 ICMP TS	Stimulus-Response	Standard	Both
Test-14 ICMP Mask	Stimulus-Response	Standard	Both
Test-20 SynEcn	Stimulus-Response	Standard	Active
Test-21 no flags	Stimulus-Response	Non-standard	Active
Test-22 SynFinUrgPsh	Stimulus-Response	Non-standard	Active
Test-23 Ack open	Stimulus-Response	Standard	Active
Test-24 Ack closed	Stimulus-Response	Standard	Active
Test-25 FinUrgPsh	Stimulus-Response	Standard	Active
Test-26 Echo Request	Single Packet	Standard	Passive

2.2.1.1 Type

Following [2], we consider three types of tests: *single-packet*, *stimulus-response*, and *sample*.

Single-Packet: A single-packet test is a test requiring only a single packet, generated by the target, in order to proceed with the deduction process. This single packet is not related to any other.

Stimulus-Response: A stimulus-response test is a test where two packets are considered: the stimulus and the response. The stimulus is generated by a third party while the response is generated by the targeted host. When analyzing the response, we can correlate it with elements of the stimulus to extract more information. Note that all stimulus-response tests can be viewed as single-packet tests (considering only the response), but, this will result in some loss of information (see Example 2.3).

Sample: A sample test is a test requiring several similar packets sent from the target in order to observe a tendency over time (e.g., how a counter is updated).

Example 2.3 (stimulus-response tests regarded as single-packet tests) Consider a simplified version of the TCP RstAck stimulus-response test (see Test-9 in Definition A.9) where we only inspect the "don't fragment" (DF) bit (which can be either set of not) of both the stimulus and the response. Windows 2000 sp4 will not set the DF bit of the response, no matter the status of the DF bit in the stimulus. Open BSD will set the DF bit of the response, again no matter the status of the DF bit in the stimulus. MacOS 9.0, on the other hand, will use the same DF bit status of the stimulus in the response. If we get an instance of that test where the stimulus has the DF bit set and the response hasn't, then we conclude it is Windows 2000 sp4. If we were to consider this test as a single-packet test (i.e., we consider only the response packet), we would lose information. For instance, if we get a response packet with the DF bit not set, it can be either Windows 2000 sp4 or MacOS 9.0. $\diamondsuit$

2.2.1.2 Nature

The nature of a test describes the traffic generated by that test. A test is either standard or non-standard.

Standard: A test is standard if it relies entirely on semantically correct packets, i.e., packets that are part of normal network communication.

Non-standard: A test is non-standard if it relies on at least one semantically malformed packet, i.e., a packet that should not appear on a network, see Example 2.4. Note that a malformed packet is not necessarily harmful, it is simply unusual. In fact, all the tests in Table 2.1 only generate harmless packets.

Example 2.4 (non-standard test)
Test-21 analyzes the response of the target to a TCP packet with no flags. According to RFC 793, every TCP packet should have at least one flag and there is no semantics associated with a TCP packet with no flags. Thus, nothing specifies how the target should respond in such a situation. ♢

2.2.1.3 Execution Mode

A test can be executed passively, actively, or both.

Purely Passive: A test is purely passive if it relies on spontaneous events, i.e., events that cannot be triggered on-demand. For instance, Test-1 relies on the first packet of a TCP handshake (SYN) and it is not possible, in general, to trigger such a packet using a stimulus packet.

Purely Active: A test is purely active if it relies on a reactive event (the response to a stimulus) that should not occur in a network (usually the stimulus is a malformed packet). As a consequence, it is not possible (or at least very unlikely) to observe this event passively. Example 2.4 describes such an active test.

Passive and Active: Tests that are both active and passive rely on a reactive event for which the stimulus can be seen as part of normal traffic. As a consequence,

the event can be seen passively and can also be triggered actively. For instance, Test-8 considers a SYN/ACK packet in response to a SYN request. This request naturally happens quite often in a network, but it can also be injected on-demand.

When we talk about a passive (resp. active) test, we mean a test that is either purely passive (resp. purely active) or that is both passive and active.

Current OSD tools are classified based on the execution mode of their tests. Passive tools rely entirely on passive tests while active tools uses only active tests. The following two sections discuss passive and active OSD tools respectively; studying, among other things, their limitations.

2.3 Passive Approach

In passive OS discovery one is only allowed to listen to the network and deduce some information from the recorded packets. In particular, one does not probe a machine to check how it reacts in a specific situation. From the sometimes incomplete information gathered, one has to deduce the OS running on the machine. An example of a passive test is presented in Example 2.5.

Example 2.5 (passive test ARP Request)
When capturing the network traffic, if one sees an ARP request from a machine with IP address I in which the destination MAC address is set to FF:FF:FF:FF:FF:FF, then one can conclude that I is running either SunOS or MacOS prior to version 10. If the field contains random uninitialized data, then one can conclude that I is running FreeBSD 4.6, 4.6.2, 4.7, 4.8 or 5.0. All other OSes initialize the field to 00:00:00:00:00:00. This corresponds to Test-2 of Definition A.2. ♢

2.3.1 Passive Tools

A few examples of passive tools for OSD are: SinFP [5], p0f [81], Siphon [72], and ettercap [58]. Below we explain how p0f works (other passive tools work in a very similar way). This will allow us to better understand the limitations of passive tools.

2.3.1.1 p0f

We consider p0f version 2.0.8 here. p0f is one of the most popular passive OSD tools available. It offers four operating modes: Syn, SynAck, RstAck and StrayAck, the latter being experimental. Each mode only considers a single type of traffic. The Syn mode considers only SYN packets sent by the target; this corresponds to Test-1 of Definition A.1. The SynAck mode only analyzes how the target responds to a SYN packet sent to an open port; this corresponds to Test-8 of Definition A.8. The RstAck mode only analyzes how the target responds to a SYN packet sent to a closed port; this corresponds to Test-9 of Definition A.9. Finally, the StrayAck mode is experimental; it considers the TCP packets exchanged during the session (as opposed to during the initialization of the session); we have no corresponding test for that as it is not clear if this type of traffic can effectively be used to fingerprint OSes.

p0f only considers TCP traffic, and when analyzing a packet, it is interested in the following fields:

- IP packet size
- IP DF bit
- IP TTL
- TCP window size (WIN)
- TCP options

The engine of p0f is a simple string matching algorithm. Figure 2.3 provides a small peek at the p0f signature file (aka fingerprint file) for the Syn mode. The signature file contains the mapping between a specific network behavior and the corresponding OSes. Each entry is of the form

$$W : T : D : S : O : Q : OS : Details \tag{2.1}$$

where:

W: The WIN value.

T: The TTL value.

D: The DF bit (0/1).

S: The IP packet size.

O: The TCP options (in a comma separated list preserving the order they appear in the packet).

Q: Oddities about the packet (e.g., IP ID value of 0, non-zero urgent pointer, non-zero acknowledgement number).

OS: The operating system family (e.g., Windows, Linux, Mac).

Details: The OS version (e.g., 2000 SP4 for Windows, 5.1 for FreeBSD, kernel 2.2 for Linux).

Each SYN packet is transformed to a string representation (the first 6 fields mentioned above) and then matched against the signature file. p0f returns the OS contained in the first entry that matches the packet, thus every OS having the same behavior must be included in the same entry (see first entry in Figure 2.3). This makes it harder to modify the fingerprint file, especially if two OSes of different families behave identically. For instance, assume FreeBSD 4.6 and NetBSD 1.3 behave identically, then what should go in the OS field (OS family) for their signature? FreeBSD or NetBSD (see the last entry in Figure 2.3 for an ad hoc workaround).

```
%8192:128:1:48:M*,N,N,S:.:Windows:2000 SP2+, XP SP1 (seldom 98)
32767:64:1:60:M16396,S,T,N,W0:.:Linux:2.4
32768:64:1:60:M*,N,W0,N,N,T:.:FreeBSD:4.8-5.1 (or MacOS X 10.2-10.3)
```

Figure 2.3: p0f Signature File Format

Some more general features of p0f are:

- Fuzzy matching. When no perfect match is found for a given packet, fuzzy matching can be used. This is used for the TTL when the fingerprinted host is on different network segments (the packet has to go through a router). A TTL

of 100 can be interpreted as 128 or 255. As far as we know, TTL is the only use of fuzzy matching.

- Input/output. p0f can operate both on-line, on real traffic, and off-line, on a previously captured trace file. In both cases, p0f tries to provide the user with a guess for every packet. Note that the user has no direct access to the knowledge of p0f (i.e., which OSes are possible and which ones are impossible), this knowledge must be inferred from p0f's guesses.

- Separate signature files. p0f has a signature file for each operation mode and the signatures can be updated without modifying the engine. However, due to the basic rule-matching algorithm (first match only), adding a new rule requires a thorough analysis of the signature file to make sure the new rule does not conflict with an existing one.

One of the first things we notice when using p0f is that the operation modes are totally independent and do not work together at all. So starting p0f in Syn mode will only allow us to analyze the SYN packet sent by the target. To analyze different types of traffic with p0f, one must start several processes (maybe on different computers) each in a different mode. Then, the task of combining the information gathered by those processes is left to the user. This big limitation is a direct consequence of the poor knowledge representation used in p0f: it is not possible to combine knowledge from different sources.

Another interesting point with p0f is that it is single-packet based. p0f treats every packet individually as if it was the only packet available. This has two major implications. First, p0f is stateless; for each packet it will guess the OS without considering the packets previously seen (and the information they convey). Second, in SynAck and RstAck modes, p0f cannot correlate the response with its stimulus; each response is analyzed individually. This is a limitation, since, for some OSes, a specific field in the response may depend on the same field in the stimulus, see Example 2.3.

2.3.1.2 Other Passive Tools

Other passive tools are quite similar to p0f. The difference is usually at the signature level: what type of traffic they consider (TCP vs ICMP) and the information they extract from packets (some rely on TCP options while others don't).

2.3.2 Advantages and Limitations

The principal advantage of the passive approach is its non-intrusive nature: a passive tool does not need to inject packets in the network, it only analyzes the communication that occurs naturally.

Current passive tools suffer from three main problems (they are detailed below): information unavailability, restriction to single-packet rules, and lack of memory. While the first one is intrinsic to the passive approach, the other two are related to the implementation[1]. These limitations are all part of the reason why passive tools are inaccurate (see Chapter 3).

2.3.2.1 Information Unavailability

The fundamental problem of the passive approach is that the information may not be available when needed. For instance, with passive OS discovery, the system will only see packets generated as part of valid communication sequences and those communication sequences must be triggered by a third party. Moreover, it is usually the case that less information can be deduced from usual (well-formed) communication than from carefully engineered (and possibly malformed) stimulus-response sequences.

2.3.2.2 Restriction to Single-Packet Rules

All passive OSD tools that we are aware of use rules always containing a single packet. That is, for each packet they generate, from scratch, the set of OSes that could possibly create such a packet. This is a limitation in two aspects.

First, many phenomena that could help identifying artifacts specific to an operating system (or a family) cannot be described using a single packet. This is the case

[1]Every passive OSD tools that we are aware of have these limitations

with the ARP Request Retransmission test (see Definition A.6), or any retransmission test for that matter, where the goal is to monitor the delay between retransmissions of the same request.

Second, many phenomena may be represented by a single packet but at the cost of losing precious information. This is the case of stimulus-response tests. See Example 2.3

2.3.2.3 Lack of Memory

Finally, all existing passive tools are stateless, i.e., they do not have a memory. For each packet they analyze, they guess the OS corresponding to that packet, regardless of the information they have deduced beforehand. For instance, assume the first packet seen allows to deduce that the OS is part of the Windows family. However, after analyzing the second packet, the tool could propose that the OS is Linux, even if this is not possible according to the previous packet.

This implementation choice can be restrictive for someone wanting continuous monitoring of the network and greatly limits the ability of passive tools to detect network events such as IP spoofing, reboot or the presence of Network Address Translation (NAT) devices. Moreover, it greatly limits the ability of the tools to accurately identify the operating system.

2.4 Active Approach

In active OSD, one can directly probe a machine to deduce its operating system, depending on the reaction of the target to the synthesized stimuli. For instance, one can initiate a TCP handshake (i.e., with a SYN) and analyze the way the target will respond (e.g., what value is used as TTL in the SYN/ACK), see also Example 2.6. Another possibility is to stimulate the target with a malformed packet and analyze how it will behave (see Example 2.7).

Example 2.6 (standard active test)
By sending a SYN packet to a closed port of a given machine, we can get a RST/ACK packet from that machine and correlate the response with the stimulus. For instance,

some versions of MacOS will set the DF bit of the RST/ACK packet to the same value as the DF bit in the corresponding SYN packet; SunOS will set it to *Yes*; and Windows to *No*. This corresponds to Test-9 of Definition A.9. ♢

Example 2.7 (non-standard active test)
Consider what happens if we send a TCP packet with the SYN, FIN, URG, and PSH flags set. This packet is malformed as it simultaneously requests the initialization (SYN) and the termination (FIN) of a session. Since the packet is malformed, the TCP specification does not dictate how the receiver should respond. Thus, OS constructors have the freedom to implement the behavior of their choice. However, the response of each OS will be deterministic according to its specific TCP/IP stack implementation. In our particular SYN/FIN/URG/PSH example, Linux Debian 2.0 replies with a SYN/ACK/FIN packet while all versions of Windows reply with a SYN/ACK and Linux FedoraCore 1 simply ignores the bogus request. This corresponds to Test-22 of Definition A.17. ♢

Another kind of active test consists in placing the target machine in extreme conditions and monitoring how it behaves. However, this sometimes results in a disruption of the normal network activities (e.g., crashing a computer). [80] describes such a test called *SynFlood Resistance* where one sends many new SYN packets until the target's stack is full with half open connections and it cannot respond anymore to the new SYN packets. The number of SYN packet required to fill the stack provides information concerning the OS. Since these tests are disruptive, we do not consider them here.

2.4.1 Active Tools

Nmap [79] and Xprobe [3] are well known active OSD tools. Another effort in active OSD comes from Core Security Technologies where they use neural networks instead of rule matching. Unfortunately, their product is commercial, thus not much information is available. It appears that their tool is very closely related to Nmap (it uses the same tests). Thus it should suffer from the same problems as most active tools. Nmap and Xprobe are detailed below.

2.4.1.1 Nmap

Nmap (network mapper) is one of the most popular network tools. It is very versatile (port scanning, application discovery, host discovery, etc.), thus it does not focus entirely on OS discovery. Here we study only the OS discovery component of Nmap version 4.75.

Nmap considers TCP and ICMP packets. It relies on the following 11 tests (see Appendix A): Test-8, Test-9, Test-10, Test-11, Test-20, Test-21, Test-22, Test-23, Test-24, and Test-25. Some tests are sometimes executed more than once, with slightly different stimuli (e.g., different window sizes). Only Test-20 seems to be optional (all other tests are executed at least once). However it is not clear in which situation Test-20 will be executed nor in which situation a specific test will be executed more than once[2].

Due to the nature of its tests, Nmap must know about one open and one closed TCP port to provide accurate results. For that reason, Nmap normally starts with a port scan. Although the port scan can be parameterized by the user (to scan only specific port ranges), the scan usually results in the injection of several packets (up to 2,000). Nmap also makes sure the scanned host is up by sending an Echo Request (ping). Ignoring the port scan and the host check, Nmap injects a minimum of 9 packets in the network and can sometimes inject close to 100 packets. In average, we observed that Nmap sends around 30 packets.

Moreover, Nmap relies on two non-standard tests (Test-21 and Test-22). Thus it will inject at least two, but sometimes up to 40 (see Footnote 2), malformed packets.

Figure 2.4 shows an entry of the Nmap signature file. It is interpreted in the following way:

- The entry is for Windows 2000, XP and Server 2003 as mentioned in the upper section of Figure 2.4.

- Each line of the lower part corresponds to a test: it provides the response of the current OSes with respect to that test. For instance, the line starting with

[2]In theory, a test can be executed more than once, each time with different parameters for the stimulus. However, it is not clear how Nmap decides when a test should be executed twice.

T1 corresponds to Test-20 (see Definition A.15). It states, among other things, that the OSes considered here should respond to Test-20 with the DF bit set (DF=Y), a sequence number of 0 (S=0), an acknowledgement number of 1 plus the initial sequence number (A=S+), and uses the flags ACK or SYNACK (F=A|AS).

New signatures can be added to the file, however it requires a great deal of effort to gather all the information and to understand the syntax and all the subtleties. Nmap can handle duplicate entries (unlike p0f).

```
Fingerprint Microsoft Windows 2000 SP2 - SP4, Windows XP SP2, or Windows Server 2003 SP0 - SP2
Class Microsoft - Windows - XP - general purpose
Class Microsoft - Windows - 2003 - general purpose
SEQ(SP=F0-10C%GCD=¡7%ISR=FB-111%TI=I%II=I%SS=S%TS=0)
OPS(O1=NNT11|M5B4NW0NNT00NNS%O2=NNT11|M5B4NW0NNT00NNS
      %O3=NNT11|M5B4NW0NNT00%O4=NNT11|M5B4NW0NNT00NNS
      %O5=NNT11|M5B4NW0NNT00NNS%O6=NNT11|M5B4NNT00NNS)
WIN(W1=4470%W2=41A0%W3=4100%W4=40E8%W5=40E8%W6=402E)
ECN(R=Y%DF=Y%T=80%TG=80%W=4470%O=|M5B4NW0NNS%CC=N%Q=)
T1(R=Y%DF=Y%T=80%TG=80%S=O%A=S+%F=A|AS%RD=0%Q=)
T2(R=Y%DF=N%T=80%TG=80%W=0%S=Z%A=S%F=AR%O=%RD=0%Q=)
T3(R=Y%DF=Y%T=80%TG=80%W=402E%S=O%A=O|S+%F=A|AS
      %O=NNT11|M5B4NW0NNT00NNS%RD=0%Q=)
T4(R=Y%DF=N%T=80%TG=80%W=0%S=A%A=O%F=R%O=%RD=0%Q=)
T5(R=Y%DF=N%T=80%TG=80%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
T6(R=Y%DF=N%T=80%TG=80%W=0%S=A%A=O%F=R%O=%RD=0%Q=)
T7(R=Y%DF=N%T=80%TG=80%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
U1(DF=N%T=80%TG=80%TOS=0%IPL=38%UN=0%RIPL=G%RID=G%RIPCK=G
      %RUCK=G%RUL=G%RUD=G)
IE(DFI=S%T=80%TG=80%TOSI=Z%CD=Z%SI=S%DLI=S)
```

Figure 2.4: Nmap Signature File Format

There are two major drawbacks to the signatures of Nmap:

- Since each entry contains the result for all the tests, there is a lot of duplication. Each time an OS can provide at least two fundamentally distinct behaviors with respect to a test[3], the whole entry has to be duplicated. Since each entry is quite big, it leads to a very large signature file (it currently contains more than 25,000 lines) with a high level of duplication (there are 36 entries related to Windows 2000). It is thus quite difficult to navigate the file.

[3]Since Nmap sometimes uses several different stimuli for a given test, the signature file reflects that by saying an OS can answer in several different ways to a given test.

- Based on the tests used by Nmap, and most likely on the fields it considers when analyzing the packets, it turns out that most OS families are grouped together in the entries. For instance, most entries related to Windows 2000 also contain Windows XP and Windows server 2003. This limits the accuracy of Nmap in pinpointing the actual OS.

In order for a signature to match (and thus for Nmap to provide an answer to the user), all the fields of that signature must be observed. The only way to make this possible is to perform all the tests. This is the reason why Nmap always executes all of its tests. This is also the main reason why Nmap is so unreliable when it does not know about one open and one closed port (when no port scan is done). In that situation, some tests will not have any results and signatures are less likely to match, because some fields are not observed.

2.4.1.2 Xprobe

Another popular active OSD tool is Xprobe. We study version 2.0.3 here. Xprobe is based on seven tests (Test-8, Test-9, Test-10, Test-11, Test-12, Test-13, and Test-14) using ICMP and TCP. Each test is executed once, except Test-11 which is executed twice (with different TOS, DF and ICMP code values) and Test-12 (the only non-standard test used by Xprobe) which is almost never used. From our experiments, it seems that Test-12 must be requested by the user, although Xprobe's documentation affirms it is always performed.

Since Xprobe has only two TCP tests, the port scan is optional and not performed by default. It will guess open/closed ports and can handle, without losing too much accuracy, the case where the two ports tried have the same status.

The rule format of Xprobe (see Figure 2.5) is similar to, but more intuitive than the one of Nmap. Each entry corresponds to a single OS. For each entry, each test (called "module" here) is associated with the response provided by the corresponding OS. For instance, Figure 2.5 shows the entry for Windows 2000 SP2. Module A corresponds to test Test-11 and we see that Windows 2000 SP2 sets the DF bit and uses a TTL no greater than 128 (in fact, it uses a TTL of exactly 128, but this value could be decreased by routers) for that test. Xprobe can handle two identical entries

(two different OSes with the exact same signature) without any problem.

```
fingerprint {
OS_ID = "Microsoft Windows 2000 Workstation SP2"
#Module A
icmp_echo_reply = y
icmp_echo_code = 0
icmp_echo_ip_id = !0
icmp_echo_tos_bits = 0
icmp_echo_df_bit = 1
icmp_echo_reply_ttl = ¡ 128
#Module F [TCP SYN/ACK Module]
#IP header of the TCP SYN/ACK
tcp_syn_ack_tos = 0
tcp_syn_ack_df = 1
tcp_syn_ack_ip_id = !0
tcp_syn_ack_ttl = ¡128
#Information from the TCP header
tcp_syn_ack_ack = 1
tcp_syn_ack_window_size = 17520
tcp_syn_ack_options_order = "MSS NOP WSCALE NOP NOP TIMESTAMP NOP NOP SACK"
tcp_syn_ack_wscale = 0
tcp_syn_ack_tsval = 0
tcp_syn_ack_tsecr = 0
...
}
```

Figure 2.5: Xprobe Signature File Format

From the signature of Figure 2.5, we can observe a serious limitation of Xprobe: it does not use the stimulus information for correlation. For instance, Xprobe does not use the fact that a specific OS (e.g. FreeBSD 4.0) will echo the DF bit value for Test-11. This means that when stimulated with an echo request where the DF bit is set (resp. not set), FreeBSD 4.0 will respond with an Echo reply where the DF bit is set (resp. not set).

The signature file of Xprobe is quite easy to understand and very easy to modify. Once a signature is added to the file, it is automatically considered by the engine. Xprobe also gives the user the possibility of developing his own fingerprinting modules (tests), but we won't discuss this further.

Xprobe uses a fuzzy matching method based on Optical Character Recognition (OCR) [43] instead of using a simple string matching. It is similar to Nmap in the sense that it first executes all tests and then consults the fingerprint database. However, Xprobe assigns a score to each individual test (based on the results obtained from the scan) and then select the signature with the best resulting score. The score of a signature is computed by summing the scores of its corresponding tests. Xprobe then

outputs the operating system(s) corresponding to the selected signature. This fuzzy matching helps Xprobe avoid a costly port scan.

2.4.2 Advantages and Limitations

The main advantage of the active approach is the ability to get information on request (whenever it is needed). Another advantage is the possibility to obtain high quality information, i.e., usually we can extract more information from the response to a carefully crafted (and possibly malformed) stimulus than from usual communication.

The main limitation of the active approach is its intrusive nature. There are two main reasons why active tools are intrusive: they generate a lot of traffic, and some of this traffic can be malformed.

2.4.2.1 Amount of Traffic Generated

The main problem with active OS discovery is the large amount of traffic generated in order to discover the OS. Below we discuss three reasons why active tools generate so much traffic.

First, some active tools need to know about one open and/or one closed port on the target to perform the tests. Nmap, for instance, first performs a port scan. A port scan by itself can sometimes generate more than a thousand network packets. It is usually possible to disable the port, but this often results in a significant decrease of accuracy. In theory, with continuous passive monitoring of the network, one could deduce the state of some ports and thus avoid the port scan.

Second, since there does not exist a single test such that one can be sure to learn the exact OS, it is necessary to come up with a sequence of tests in order to correctly determine the operating system. This gives rise to multiple sequences of actions that may all lead to achieving the goal. Most active tools don't resort to planning and simply execute all available tests. Furthermore, active tools are designed to answer questions of the form "Which OS is running on a given machine?". To answer effectively (i.e. without doing all the work to know the exact OS) a less restrictive (but not less interesting) question such as "Is a given machine running the operating system *o*?", an active tool would have to reason to generate a judicious sequence of

actions; such a feature is not available in today's active tools.

Finally, another problem with active tools comes from the lack of continuous monitoring. An active tool executes the tests, gives the result and then shuts down until the next query. When the next query comes in, the active tool must do all the work again (i.e. run all tests again), even if the query is the same. This is unacceptable in a situation where a third party tool rely on the information provided by an OSD tool (i.e., the same queries will be repeated over and over again), especially if the amount of traffic generated needs to be minimized.

2.4.2.2 Malformed Traffic

Another major drawback of active tools is the injection of malformed packets in the network. The use of malformed packets is justified by the fact that one can usually glean more information from the response to a malformed request (i.e. a request that should never occur). Since these requests are malformed, there is no standard way to answer them and chances are that different OSes will handle them quite differently (e.g. responding as if they were a valid request, or responding with an error message, or not responding at all, etc.). This malformed traffic becomes a big problem when it interferes with other network equipments, for instance an intrusion detection system (e.g., [77] reports cases in which the Cisco IDS will detect Nmap fingerprinting a host and will generate an alarm). Using planning could help avoiding, or limiting as much as possible, the injection of malformed packets.

2.5 Obfuscating OS Discovery

Initially, OS discovery was directly associated with hacking. Indeed, hackers often need to obtain target information before executing an attack. To prevent this information leak, techniques were developed to hide the idiosyncracies of OSes. Two approaches have been proposed [36]: host-based and network-based.

The Host-based method consists of directly modifying the TCP/IP stack implementation of a host so it does not exhibit the expected behavior of its OS. For instance, we can modify the stack of a Windows machine to make it look like the

more secure OpenBSD OS. However, such modifications are error-prone[4] and difficult. Moreover, they must be applied to every host on the network. For those reasons, it is recommended [36] not to adopt a host-based OS fingerprint obfuscation method. A host-based modification would disrupt the ability to perform OS discovery.

The network-based approach [70] consists of adding a traffic scrubber device at the entry point of a network. This device modify the egress traffic making every computer in the network look identical, thus preventing OS discovery from outside. The main advantage of this technique is that it handles every computer in the network at once. Moreover, it does not prevent OS discovery from within the network (e.g., performed by the network administrator), because the traffic inside the network remains unchanged and exhibits differences in TCP/IP stack implementations.

In conclusion, host-based techniques to obfuscate OS fingerprints would prevent OS discovery, but their use is proscribed (both for their difficulty to deploy and the risks they incur). Network-based obfuscation techniques, one the other hand, still allow OS discovery from inside the network and are much safer, easier, and faster to deploy.

2.6 Discussion

Table 2.2: Advantages/Inconvenients of OSD Approaches/Tools

OSD Tools	**Advantages**	**Limitations**
Passive	• Access to purely passive tests • Non-intrusive	• No access to purely active tests • No access to information on demand • No memory • No continuous monitoring • Single-packet analysis • Ad hoc and informal
Active	• Access to information on demand • Access to purely active tests	• Oriented toward a single query • No continuous monitoring • Intrusive • Ad hoc and informal

The advantages and limitations of the two OSD approaches (passive and active)

[4] According to [36], modifying the stack implementation could prevent some application from running correctly.

seem complementary, see Table 2.2. For instance, some phenomena can only be observed passively, while others can only be observed actively. Hence, it seems possible to combine the two approaches together and obtain a better tool.

Moreover, current tools are quite simplistic, leading to some drawbacks, e.g., the inability of passive tool to consider stimulus-response tests and the lack of reasoning for test selection in active tools.

We believe there is a lot of room for improvements. But before designing our own tool, it is important to verify if the accuracy of current OSD tools truly suffers from the limitations discussed in this chapter. This verification will be done through an experiment presented in the next chapter.

Chapter 3

Motivation

Good is not good enough when better is expected.
-Lou Lamoriello

This chapter provides our motivations for designing a new OS discovery approach. First, Section 3.1 argues why OS discovery is useful. Then, Section 3.2 illustrates the inaccuracy of current OSD tools through an experiment.

3.1 Why is OSD important?

Knowledge about the operating systems of a group of computers is useful in numerous security and management-related tasks. We have classified these tasks into three categories based on the type of information they require. Our classification is based on the query a user would make to the OSD tool in order to obtain the knowledge appropriate for the task. The three queries are:

- *Single OS Query*: "Is the computer running the specific OS o?"
- *Group OSes Query*: "Is the computer running an OS belonging to the given set of OSes O?"
- *Exact OS Query*: "Which OS is running on the computer?"

Below we provide examples of tasks for which OS discovery is useful. The examples are grouped according to the above queries.

3.1.1 Single OS Query

The Single OS Query is interesting in a situation where a new update is available for a specific OS, and the network administrator needs to find all the computers

that must be updated. For instance, when service pack 3 for Windows XP was released, administrators had to find all the computers running Windows XP sp2 to update them. The task is to check, for each computer, whether it runs Windows XP sp2. This task can be performed by using the Single OS Query (where o represents Windows XP sp2).

3.1.2 Group OS Query

The Group OS Query is very important from a security point of view.

The task of making sure that a network respects the company's policies regarding which operating systems are allowed can be modeled using this query. Simply verify if each computer is running an OS which belongs to the set of permitted OSes.

When a new vulnerability is released, a set of vulnerable products is usually provided. Often, these products are OSes. Thus, it is possible for a network administrator to check if his network is affected by a new vulnerability using the Group OS Query. This is done by checking, for each computer, if its OS is part of the vulnerable products.

The most important task related to this query is probably the automatic filtering of *non-critical* IDS alarms. The idea is to filter out IDS alarms related to an attack that should fail because the target is not vulnerable to this attack. This is done by testing if the target operating system belongs to the set of OSes that are vulnerable to the ongoing attack. More detail about this task is presented in Section 3.2.3.

3.1.3 Exact OS Query

The Exact OS Query (i.e., finding the actual OS running on a computer) is the focus of current OS discovery tools. It is useful in building a network inventory. Having an accurate and up-to-date network inventory is definitely important. It can help prevent breakdowns caused by the deployment of new software. It can also help reduce the cost of licensing; if no computer is running Sun OS anymore, the company has no need to pay for support and updates.

3.2 Evaluating Current OSD Tools

Now that we have identified several tasks where it is important to have information about operating systems, we can investigate whether current OSD tools are sufficiently reliable to perform those tasks. We will do this by evaluating OSD tools with respect to two queries: Group OS Query (Section 3.2.3) and Exact OS Query (Section 3.2.4). Both queries will be evaluated on the same dataset, which is described below.

3.2.1 Dataset

Evaluating OS discovery tools is not trivial. Passive tools require traffic to be generated, either on-line or off-line (using recorded traffic traces). Active tools, on the other hand, have to be handled on-line as they need to communicate with the actual computers. Therefore, to evaluate OSD tools, we need a set of traffic traces (or a strategy to generate traffic) and we need access to the computers used to generate the traffic.

We used the vlab infrastructure developed by the Communications Research Centre Canada as our evaluation environment. For the passive tools, we used the publicly available intrusion dataset [51] generated with vlab (see Appendix B for more information on the dataset). This dataset contains 6,656 traces. For the active tools, we had direct access to the machines used to generate the intrusion dataset.

The vlab infrastructure has 95 machines, each corresponding to a different OS (34 versions of BSD, 25 of Linux, and 36 of Windows) and with many different applications.

Although we conducted our experiment on a single dataset, we believe our results to be reliable, mainly due to the important diversity of the dataset. The dataset might not reflect real networks, but this allows us to evaluate the tools/approaches in general, without the risk of having idiosyncracies guiding the results. Moreover, having several OSes form the same family (e.g., 34 BSD) allows us to evaluate OSD tool in difficult conditions. Indeed, it is much harder to distinguish FreeBSD 6.0 from FreeBSD 6.1 (being from the same family, they will have similar TCP/IP stacks and thus similar behavior) than from Windows XP (which has a completely different TCP/IP stack).

Adequate datasets are quite hard to obtain, they are usually either normalized to prevent leaking sensitive data or simply not publicly available. It is harder still to have access to the machines used to generate the dataset.

3.2.2 OSD Tools

Our evaluation was performed using nine popular OSD tools: seven passive tools: p0f [81] in Syn, SynAck, RstAck, and StrayAck modes, SinFP [5], Siphon [72], and ettercap [58] as well as two active tools: Nmap [79] and Xprobe [3]. Note that the four modes of p0f are totally independent and cannot be used togheter; as a consequence, we consider them as four different tools.

p0f, Xprobe, and Nmap are the most popular OSD tools as evidenced by their apperance in numerous OSD papers [47, 65, 70]. Moreover, p0f and Xprobe are listed as the top 2 OSD tools in the 2006 network tools surveys performed by Nmap developpers[1]. The SANS Intitute also considers p0f, Xprobe, and Nmap as the three most popular OS fingerprinting tools [1]. We are not aware of any popular commercial tool dedicated to OS discovery. Some commercial tools have an OSD module; however, this module is usually closely related to an open source tool. For instance, Sourcefire RNA[2] has a passive OSD module inspired from p0f, while NetScanTools Pro[3] has an active OSD module very similar to Xprobe. We did not evaluate any of the commercial OSD modules because there is no indication that they are enhanced versions of their open source counterparts.

3.2.3 Group OS Query

To evaluate OSD tools with respect to the Group OS Query, we measure their ability to provide contextual information in order to filter out non-critical IDS alarms. More information about this experiment can be found in [24, 25] and detailed experimental results can be obtained from `http://hosd.sourceforge.net/experiments/2009-04.html`.

[1] `http://sectools.org/os-detectors.html`
[2] `http://www.sourcefire.com`
[3] `http://www.netscantools.com`

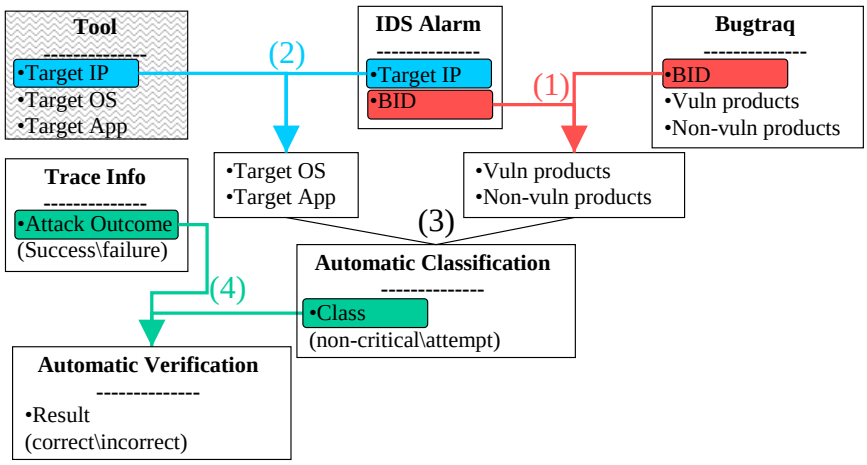


Figure 3.1: Automatic Evaluation Process - OSD Tools

The experiment works as follows (see Figure 3.1). We feed each trace to an IDS and record the alarm related to the attack attempt. Then, for a specific alarm:

(1) Based on a vulnerability repository, such as Security Focus [68], we obtain the set of OSes that are vulnerable to the attack present in the trace. The link between the alarms and an entry on Security Focus is through the Bugtraq ID (BID) in Snort's alarms.

(2) Using an OSD tool we check if the target is running a vulnerable OS (using the Group OS Query).

(3) If the target is not vulnerable, then the attack will fail and the alarm is then classified as non-critical (NC), see Figure 3.2. Otherwise the alarm is classified as an attempt (A).

(4) Since the dataset is well-document (we know, for each trace, the status of the attack, i.e., success/failure), we can automatically see if we misclassified the alarm.

To obtain an upper bound on the number of non-critical alarms that can be identifed based on the target's OS, we also run the experiment assuming a perfect OSD tool (i.e., we use the actual OS of each target as provided by the trace documentation).

ContextOS(a)

Classify a given alarm based on the vulnerability of the target OS
Input: a: the alarm
Output: The alarm classification
Notes: pOS(t) is the set of possible OSes for target t
NV(b) (resp. V(b)) is the set of non-vulnerable (resp. vulnerable) products for BID b.

```
class ← A
IF pOS(a.target) ⊆ NV(a.bid)
    class ← NC
ELSE IF pOS(a.target) ∩ V(a.bid) = ∅
    IF V(a.bid) ⊆ OS
        class ← NC
RETURN class
```

Figure 3.2: ContextOS Algorithm

This simulates a perfect OSD tool and thus provides an ideal scenario. Indeed, if a non-critical alarm cannot be identified as such when knowing the actual OS, we cannot expect to identify it as non-critical based on the information from an OSD tool.

Since most OSD tools will provide several independent guesses for a single trace (for passive tools) or a single run (for active tools), we consider that OSD tools provide a set of possible OSes. The set of possible OSes provided by a passive tool for a given trace is the union of all the OSes guessed by the tool when analyzing the given trace. The set of possible OSes provided by an active tool for a given trace is the set of OSes reported when running the tool against the computer which acts as the target in the trace.

3.2.3.1 Results

The goal is to identify the noncritical alarms among all available alarms. This can be viewed as an information retrieval task [69]. For this reason, we use the classical measures of information retrieval to assess the accuracy of OSD tools. We mainly use precision and recall:

$$Precision = \frac{\#\text{ of noncritical alarms classified as NC}}{\#\text{ of alarms classified as NC}}$$

$$Recall = \frac{\#\text{ of noncritical alarms classified as NC}}{\#\text{ of noncritical alarms}}$$

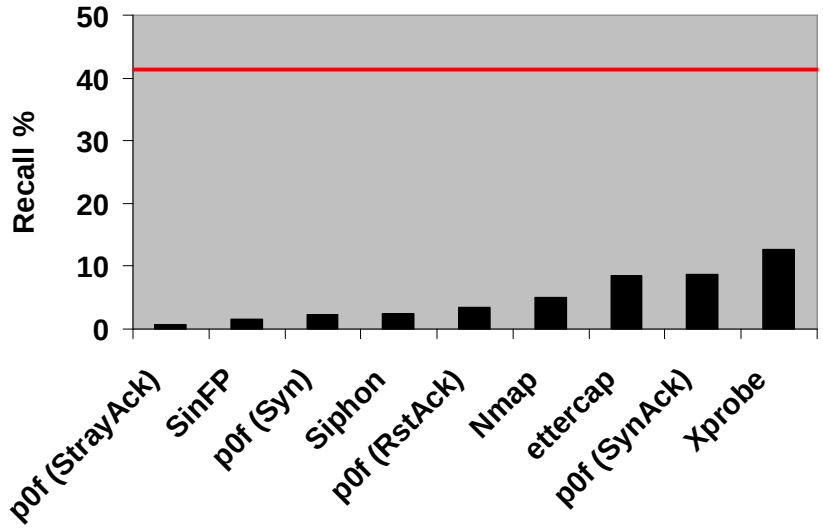


Figure 3.3: OSD Tools Recall

The perfect tool would have a recall of 100% (it is able to classify every noncritical alarm as NC) and a precision of 100% (it does not classify any critical alarm as NC). When interpreting the results, it is important to consider that a decrease in precision (i.e., critical alarms being classified as NC) is more harmful than a decrease in recall (i.e., non-critical alarms being classified as A).

Figure 3.3 presents the recall summary for the nine OSD tools. The horizontal line at 41% represents the maximum an OSD tool can achieve, i.e., the recall when knowing the exact OS of each target. The potential is not 100% because not every non-critical alarms can be identified as such simply based on the target's OS. Indeed, an attack can fail for other reasons (e.g., the packet did not reach the target, the target did not interpret the packet in the same way as the IDS). As we can see, the best tools achieve only 1/3 of the potential (13% vs 41%). This is clearly unsatisfactory and indicates a need for better OSD tools.

Figure 3.4 presents the precision summary for the nine OSD tools. The horizontal line slightly above 99% represents the precision when assuming exact knowledge of the OSes. The mistakes are due to missing entries in Security Focus. For instance, the dataset contains an exploit for BID 9633 which successfully compromised a Windows 2000 sp4 target; however, Security Focus does not list this product as being

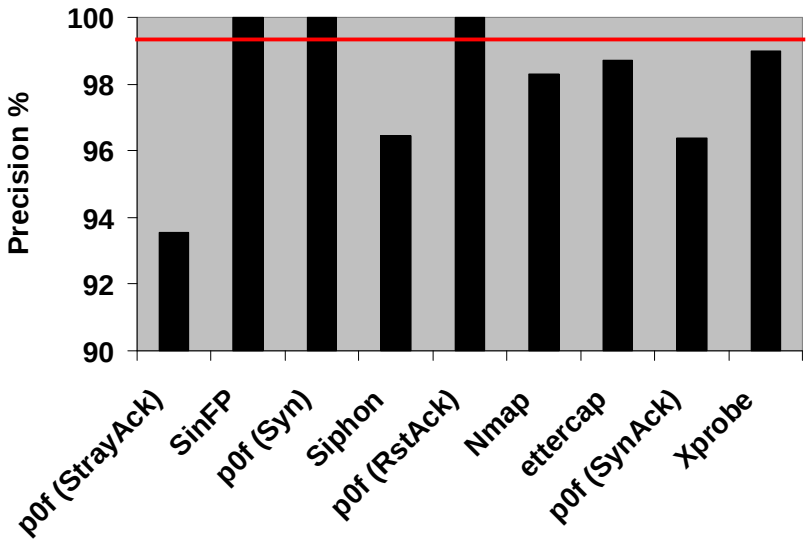


Figure 3.4: OSD Tools Precision

vulnerable.

We can see that most tools do not incur a significant loss of precision. The fact that three tools provide a precision above 99% can be explained in the following way. Consider a successful execution of an exploit for BID 9633 against Windows 2000 sp4. Now assume SinFP is not able to identify the OS in that case. Thus, this attempt will not be classified as non-critical by SinFP (which is correct). However, it was misclassified as non-critical when knowing the exact OS (due to a missing entry in SecurtyFocus, see discussion above). Hence, SinFP didn't make a mistake where the exact knowledge did.

There are two important conclusions to this experiment:

- Knowledge of the OSes running in a network is very useful for identifying non-critical IDS alarms.

- Current OSD tools are not accurate enough to obtain the knowledge required to identify non-critical IDS alarms (achieving only 1/3 or their potential).

3.2.4 Exact OS Query

In this second experiment, we want to measure the ability of OSD tools to determine the actual OS of a computer. Once again, we consider the set of possible OSes provided by a given tool for a given trace. Based on that, we say that a tool returns:

- a *correct answer*, when the set of possible OSes returned by the tool contains the actual OS.
- an *incorrect answer*, when the set of possible OSes returned by the tool does not contain the actual OS.
- an *inconclusive answer*, when the set of possible OSes returned by the tool is empty (or contains all the OSes).

It is preferable to obtain a correct answer. Otherwise, it is better for a tool to provide an inconclusive answer than an incorrect one, that way we do not take actions based on erroneous information.

When a tool provides a correct answer, it is important to look at the size of the set of possible OSes returned. Indeed, it is quite easy to always provide a correct answer simply by tagging almost every OS as possible. There is a tradeoff here which is quite similar to the recall/precision tradeoff.

3.2.4.1 Results

Even though this is not a classical information retrieval task, we are looking for the one good result in a set. Therefore, we still consider similar measures to provide intuitive comparison between different tools. First, we consider how often a tool will give us a correct answer, an incorrect answer, and an inconclusive one. In a sense, this is similar to the recall measure, but we call it correctness here to avoid confusion. Second, we consider what is the size of the possible OSes set whenever the tool provides a correct answer. This relates to the inverse of the precision measure, we call it imprecision here. The lower the imprecision, the better.

To allow for a general comparison between different tools, we provide the correctness/imprecision summary for each tool, as defined below:

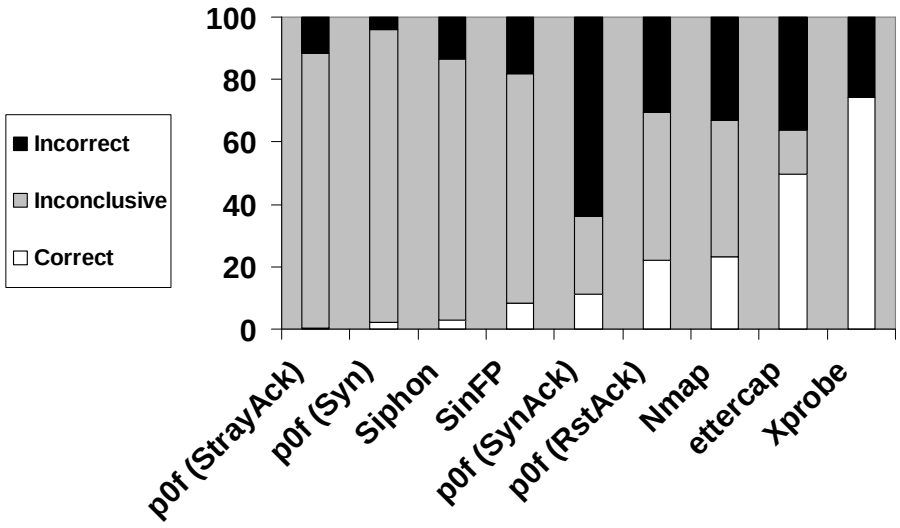


Figure 3.5: OSD Tools Correctness

- Correctness summary: percentage of traces for which the tool provided a correct answer, percentage of traces for which the tool provided an incorrect answer, and percentage of traces for which the tool returned an inconclusive answer.

- Imprecision summary: the average imprecision over all the traces for which the tool provided a correct answer (i.e., the average size of the set of possible OSes).

Ideally, a tool would have a high percentage of correct answer (the closer to 100% the better), and a low imprecision (the closer to 1 the better), i.e., it only returns a small set of possible OSes.

Figure 3.5 provides the correctness summary for the nine OSD tools. The white, grey, and black areas correspond to the percentage of correct, inconclusive, and incorrect answers respectively. The following observations are of interest:

- Most existing tools have a very poor correctness. With two exceptions, ettercap and Xprobe, they rarely identify the actual OS (less than 25% of the time).

- Most existing tools often guess wrong. The five tools with highest correctness (p0f in SynAck and RstAck mode, Nmap, ettercap and Xprobe) have provided an incorrect answer 25% of the time or more.

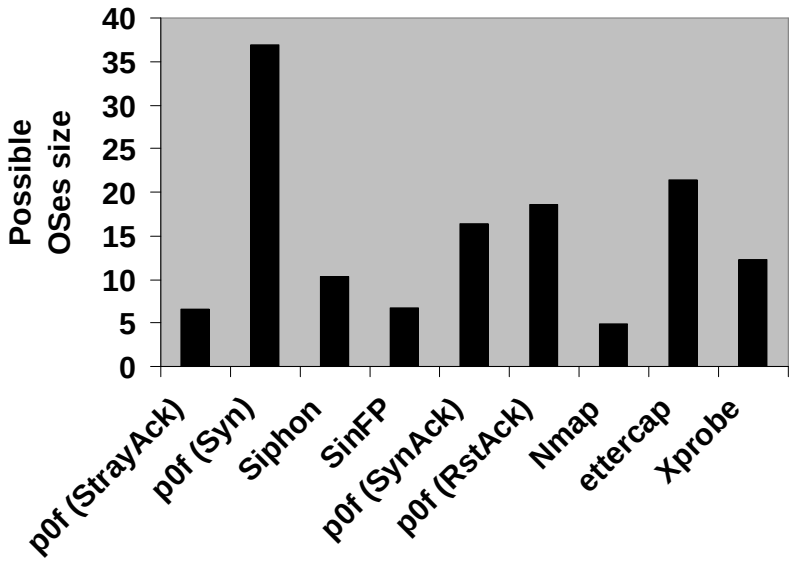


Figure 3.6: OSD Tools Imprecision

- There is no clear distinction between passive and active tools. We were expecting active tools to be much better than passive ones. Instead, ettercap (a passive tool) performs better than Nmap, an active one.

Figure 3.6 provides the imprecision summary for the nine OSD tools. We note the following:

- Nmap has a good precision. This might be an explanation for its poor correctness (see Figure 3.5): more importance being accorded to achieving a small set of possible OSes.

- There is a distinction between active and passive tools. Active tools have an average size for the set of possible OSes around 10, while most passive tools are around 20 (passive tools with a good precision all have a dramatically poor correctness).

3.3 Discussion

The results presented here confirm that the shortcomings of the current OSD approaches, as discussed in Chapter 2, do have a negative impact on their accuracy.

The absence of a memory in passive tools leads to a poor precision, they fail to rule out OSes based on previous network events. The inability to correlate a response with its stimulus also lead to a poor precision, see Example 2.3.

The poor recall/correctness is mainly due to the lask of a good knowledge management approach in current OSD tools. The actual OS is often ruled out from the set of possible OSes by an unsafe, and informal, deduction process.

In this thesis, we pursue the goal of developing a new OSD approach. Before jumping into that, the following chapter introduces the theory of diagnosis which will serve as the corner stone of our new approach. Diagnosis will help in providing a strong knowledge management core, including a memory and a safe deduction process, for our OSD tool. Moreover, the explicit test selection process in diagnosis will allow our tool to consider many OSD tests without ever needing to execute more than a few.

Chapter 4

Background - Diagnosis

Everybody is ignorant, only on different subjects.
Will Rogers

Since our proposed approach to OS discovery will rely heavily on the theory of diagnosis, mainly to provide a framework with a strong knowledge management component, this chapter provides an introduction on this topic. Section 4.1 introduces diagnosis. Sections 4.2 and 4.3 present the two important components of diagnosis: candidate generation, which will provide a safe deduction process to build the set of possible OSes, and candidate elimination, which will provide mechanisms for selecting tests to be executed. Finally, Section 4.4 discusses some limitations of the theory of diagnosis, mainly from the OS discovery point of view.

4.1 What is Diagnosis?

The goal of a diagnosis engine is to find the broken components of a system [66]. This goal can be generalized to finding the explanations for the observed behavior of a system [55].

Below we present a general view of diagnosis problem solving as well as the basic diagnosis terminology. Then we take a look at the four major types of diagnosis problems. Finally, we provide a set of properties that help to define a specific diagnosis problem.

4.1.1 Diagnosis Problem Specification

According to [66], which provides the foundations to the theory of diagnosis, a diagnosis problem is a triple $\langle$ CONST, SD, OBS$\rangle$ where:

CONST: A finite set of constants representing constituents available to build an explanation, referred to as explanatory constituents. For instance, in the medical field, these would be diseases.

OBS: The set of possible observations for the system. Based on a subset of the possible observations, we will try to determine which constituents are responsible for these observations. For instance, these would be disease symptoms.

SD: The system description provides the information for diagnosing the system, i.e., it provides a direct or indirect way of linking the observations to the constituents and then explaining the observed behavior. SD will be represented as a set of logical formulas. For instance, we could know that suffering from chicken pox causes red spots and fever, while suffering from allergies causes red spots and swollen glands.

Figure 4.1 illustrates the general behavior of a diagnosis tool (inspired from [39]). Each step is described individually below:

A: The diagnosis tool obtains some observations from the system. For instance, a doctor observes that the patient is covered with red spots.

B: The diagnosis tool then computes the possible explanations for the observations and updates its knowledge base. For instance, the doctor deduces that the patient could suffer from either chicken pox or an allergy.

C: The tool then checks if it can provide the actual diagnosis to the user.

D: If so, then the work is completed.

E: Otherwise, as in our example where the doctor has two possible explanations, more work is required before obtaining the actual diagnosis.

F: The diagnosis tool then selects a test that will generate new observations which will, hopefully, help determine the actual diagnosis. For instance, the doctor could check the glands of the patient to see whether they are swollen. The selected test is executed. This will stimulate the system and generate new observations and we go back to step A.

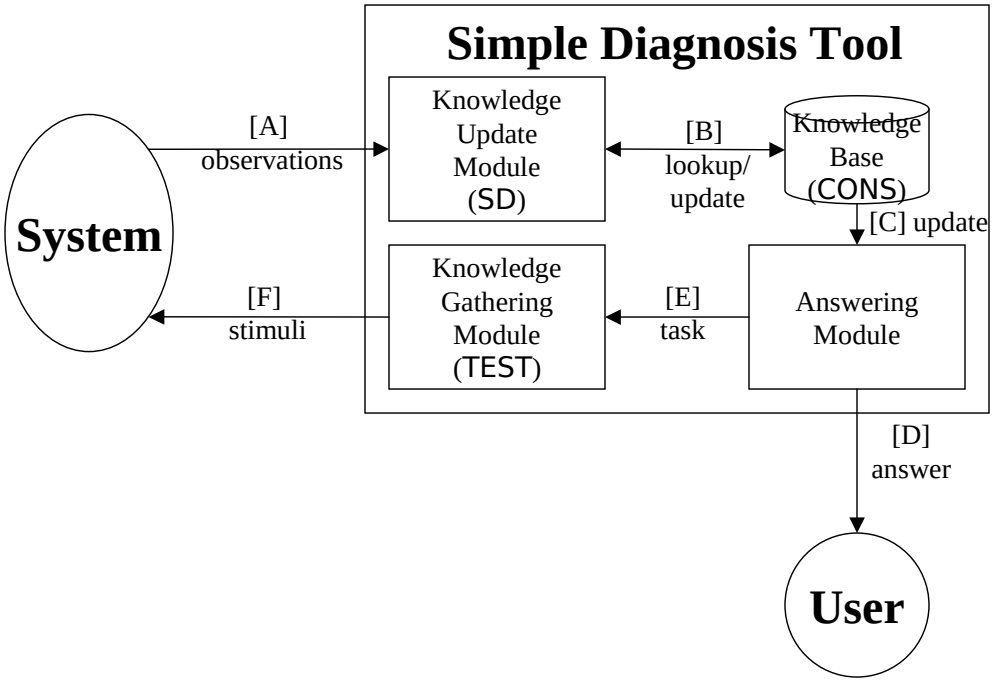


Figure 4.1: Simple Diagnosis Tool

4.1.2 Diagnosis Terminology

Three terms are of particular interest in diagnosis: the *hypothesis space*, a *diagnosis candidate* and the *actual diagnosis*. They are defined below.

Definition 4.1 (Hypothesis Space)
The hypothesis space [26], denoted $\mathcal{H}$, defines the hypotheses we can consider when trying to explain the observations made on the system. The hypothesis space is formed with the elements of $CONST$; depending on the diagnosis problem settings, we can use $\mathcal{H} = \wp(CONST)$ or $\mathcal{H} = \{\{c\}|c \in CONST\} \cup \{\emptyset\}$ (where $\emptyset$ means that the system appears to be working normally, i.e., no explanation is required). ◯

The content of $\mathcal{H}$ depends on the fault cardinality property of the underlying diagnosis problem. A diagnosis problem can consider single or multiple fault(s). In the single fault case, the given observations must be explained by at most one constituent (zero if everything is fine). In that case, $\mathcal{H} = \{\{c\}|c \in \mathsf{CONST}\} \cup \{\emptyset\}$. In the multiple faults case, the observations can be explained by the combination of

multiple explanatory constituents. In that case, $\mathcal{H} = \wp(\mathsf{CONST})$. In both cases, $\mathcal{H}$ contains sets of explanatory constituents.

Definition 4.2 (Diagnosis Candidate)
A diagnosis candidate [66] (or candidate for short) is a hypothesis from $\mathcal{H}$ explaining the given observations. For a given set of observations, there might be more than one candidate. Each candidate is a subset of CONST, and we use $\Gamma(\Theta)$ to denote the set of diagnosis candidates for observations Θ. ◯

Definition 4.3 (Actual Diagnosis)
The actual diagnosis [32] is the single hypothesis describing the actual state of the current system. ◯

There are two key processes in diagnosis: candidate generation and candidate elimination. Candidate generation consists of computing (resp. maintaining) the set of diagnosis candidates for some given (resp. new) observations (see Section 4.2). This corresponds to steps A and B in Figure 4.1. As for candidate elimination, it is responsible for the test selection process with the objective of obtaining the actual diagnosis (i.e., eliminating candidates until only one, the actual one, remains), see Section 4.3. This corresponds to steps E-F in Figure 4.1.

So far, we have kept the notion of *explaining* the observations quite vague. This was intentional because there are two accepted definitions for the concept of explanation: consistency-based [66] and abductive [61] (see below).

Definition 4.4 (Consistency-Based Explanation)
Given a diagnosis problem $\langle\ \mathsf{CONST}, \mathsf{SD}, \mathsf{OBS}\ \rangle$, a consistency-based diagnosis candidate for some given observations $\Theta \subseteq \mathsf{OBS}$ is $\Delta \in \mathcal{H}$ such that:

$$\mathsf{SD} \cup \Theta \cup \{P(c) | c \in \Delta\} \cup \{\neg P(c) | c \in \mathsf{CONST} \setminus \Delta\} \text{ is consistent.}$$

Where P is a predicate describing the status of the constituents (e.g., "broken" or "abnormal" when talking about physical component and "suffersFrom" when talking about disease). More intuitively, $\Delta \in \mathcal{H}$ is a candidate if assuming that the constituents of Δ have property P, while the others don't, is consistent with the observations. The consistency-based concept of an explanation was introduced in [66].
◯

Definition 4.5 (Abductive Explanation)
Given a diagnosis problem $\langle$ CONST, SD, OBS $\rangle$, *an abductive diagnosis candidate for some given observations* $\Theta \subseteq \mathsf{OBS}$ *is* $\Delta \in \mathcal{H}$ *such that:*

$$\mathsf{SD} \cup \{P(c) | c \in \Delta\} \text{ is consistent}$$

and

$$\mathsf{SD} \cup \{P(c) | c \in \Delta\} \models \Theta$$

More intuitively, $\Delta \in \mathcal{H}$ *is a candidate if the assumption that the constituents of* Δ *have property* P *is sufficient to predict the given observations. The abductive concept of an explanation was introduced in [61].* ◯

Abductive reasoning and consistency-based reasoning form two families of diagnosis problems (with different properties and algorithms). Another important distinction among diagnosis problems is the *level* of information contained in SD. The two trends are: model-based diagnosis vs rule-based diagnosis.

In model-based diagnosis [66], SD contains rules about the interaction of the system's components (this is viewed as "high-level" information). From that model of the system, we can compute diagnosis candidates. In rule-based diagnosis [64], SD contains rules directly associating the observations and the explanatory constituents (this is considered as "low-level" information). Note that model-based and rule-based diagnosis can use both abductive and consistency-based reasoning. This produces four families of diagnosis problems. In order to be able to determine in which family is the OS discovery problem, Section 4.1.3 briefly presents these families.

4.1.3 Four Diagnosis Families

The four families of diagnosis problems considered here are defined using two properties: the structure of SD (rule-based or model-based) and the explanation mechanism (abductive vs consistency).

The structure of SD strongly depends on the level of knowledge we extract from the underlying system. For instance, it seems quite impractical to come up with a high-level model for human medical diagnosis. In that case, the human body would have to be partitioned into components and the interaction of those components would have to

be expressed as logical formulas (an attempt was made in [60]). Thus it is more natural to represent the medical knowledge directly in the form of relations between symptoms and diseases (low-level knowledge). Other domains, like engineering diagnosis, are naturally expressed in terms of models. For instance, a circuit board can easily be broken down into components, and the interaction of those components is easy to describe in logic. Usually, model-based diagnosis is preferred to rule-based diagnosis because it is considered more formal. Indeed, most, but not necessarily all, rule-based diagnosis approaches are expert systems relying on humans to provide knowledge in an ad hoc way. In some cases however, it is simply impossible to design a model, and one has to rely on a rule-based approach.

The choice of the explanation mechanism depends on the kind of knowledge (instead of the level) we extract from the system. For model-based diagnosis, we talk about *normal* vs *abnormal* behavior knowledge (see [61]), while for rule-based diagnosis we talk about *explanatory* vs *fault-descriptive* knowledge (see Example 4.1 below). Normal behavior knowledge and explanatory knowledge require consistency-based reasoning, while abnormal behavior knowledge and fault-descriptive knowledge require abductive reasoning [63]. Example 4.1 illustrates the two cases for the rule-based situation. The work on choosing the proper explanation mechanism based on the kind of knowledge was mainly done by Poole in [61, 62, 63].

Example 4.1 (kinds of knowledge for rule-based diagnosis)
Consider rule-based medical diagnosis. We can choose to represent the rules of SD in two different ways: explanatory knowledge and fault-descriptive knowledge. In explanatory knowledge, we have rules of the form

$$\text{suffersFrom(chicken pox)} \ \vee \ \text{suffersFrom(allergy)} \ \leftarrow \ \text{red spots}$$

where each symptom is directly associated with the diseases that are the possible explanations. This kind of knowledge requires the use of consistency-based reasoning [63]. Fault-descriptive knowledge, on the other hand, associates each disease to its symptoms as follows

$$\text{suffersFrom(allergy)} \ \rightarrow \ \text{red spots} \ \wedge \ \text{swollen glands}$$

This kind of knowledge will be used with abductive reasoning. $\diamondsuit$

Below, we study some of these families in more detail. In Section 4.1.4 we present Reiter's approach to diagnosis. This approach is model-oriented and relies on consistency-based reasoning. Reiter's work contains most of the theory and algorithms in the diagnosis literature. However, model-based diagnosis does not apply well to OS discovery; we don't have a concrete system to model. Thus, we consider a rule-based approach for OSD (we will adapt Reiter's algorithms to the rule-based approach). In Section 4.1.5 we study the work by Poole regarding the different reasoning mechanisms (abductive and consistency-based) to choose the appropriate one for OSD.

4.1.4 Reiter's Model-Based Diagnosis

Here we discuss the classic approach to diagnosis proposed by Reiter in his seminal paper [66].

Reiter's approach is model-based and mainly oriented towards engineering (e.g., circuits diagnosis). The knowledge in SD is of the normal behavior type, thus consistency-based reasoning is used. Moreover, a multiple faults hypothesis space is considered.

To illustrate Reiter's diagnosis approach, we consider the simple full adder device of Example 4.2.

Example 4.2 (full adder (taken from [66]))
The device, shown in Figure 4.2 consists of three input bits (I_i), five gates (X_i "xor", O_i "or", A_i "and") and two response bits (R_i). The device will sum the three input bits to provide the bit R_1 and the carry R_2. ♢

Based on the full adder device, the diagnosis system would be:

CONST: $\{A_1, A_2, X_1, X_2, O_1\}$

OBS: $\langle I_1, I_2, I_3, R_1, R_2\rangle$ where each I_i and R_i has a binary value (0 or 1), see Example 4.3.

SD: As presented in Table 4.1, the system description contains:

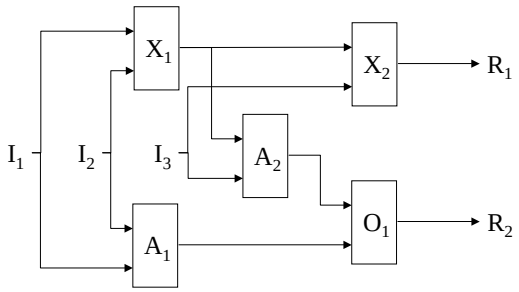


Figure 4.2: Full Adder Device

- a declaration of the components, which are also the explanatory constituents;
- the normal behavior of the components (how the components behave when they are not abnormal). Here $AB(c)$ means that component c is abnormal;
- the interaction between different components (and inputs/outputs);
- restrictions on the system input;
- and axioms for boolean algebra which are not shown here (e.g., the specification of $and(x, y)$).

Example 4.3 (full adder (continued from Example 4.2))
Suppose a physical full adder is given the inputs 1, 0, 1 and produces 1, 0 in response. This observation can be logically represented by:

$$(I_1 = 1) \wedge (I_2 = 0) \wedge (I_3 = 1) \wedge (R_1 = 1) \wedge (R_2 = 0)$$

♢

Based on the consistency-based concept presented in Definition 4.4, a diagnosis candidate here is $\Delta \in \mathcal{H}$ such that

$$\mathsf{SD} \cup \Theta \cup \{AB(c) | c \in \Delta\} \cup \{\neg AB(c) | c \in \mathsf{CONST} \setminus \Delta\} \text{ is consistent.}$$

Example 4.4 (full adder (continued from Example 4.3))
The following observations Θ for the binary full adder conflict with the expected behavior of the system.

$$\{(I_1 = 1) \wedge (I_2 = 0) \wedge (I_3 = 1) \wedge (R_1 = 1) \wedge (R_2 = 0)\}$$

Table 4.1: SD for the Full Adder

$\mathsf{ANDG}(A_1) \wedge \mathsf{ANDG}(A_2) \wedge \mathsf{XORG}(X_1) \wedge \mathsf{XORG}(X_1) \wedge \mathsf{ORG}(A_1).$
$\mathsf{ANDG}(X) \wedge \neg\mathsf{AB}(X) \rightarrow (out(X) = and(in1(X), in2(X))).$ $\mathsf{XORG}(X) \wedge \neg\mathsf{AB}(X) \rightarrow (out(X) = xor(in1(X), in2(X))).$ $\mathsf{ORG}(X) \wedge \neg\mathsf{AB}(X) \rightarrow (out(X) = or(in1(X), in2(X))).$
$(in1(X_1) = I_1) \wedge (in2(X_1) = I_2) \wedge (in1(A_1) = I_2) \wedge (in2(A_1) = I_1)$ $(in1(A_2) = out(X_1)) \wedge (in2(A_2) = I_3) \wedge (in1(X_2) = out(X_1)) \wedge (in2(X_2) = I_3)$ $(in1(O_1) = out(A_2)) \wedge (in2(O_1) = out(A_1))$ $(out(X_2) = R_1) \wedge (out(O_1) = R_2)$
$(in1(X_1) = 0) \vee (in1(X_1) = 1).$ $(in2(X_1) = 0) \vee (in2(X_1) = 1).$ $(in1(A_1) = 0) \vee (in1(A_1) = 1).$

With input $I_1 = 1, I_2 = 0, I_3 = 1$, we would expect the output $R_1 = 0, R_2 = 1$, thus the system is faulty. More formally, the system is faulty because $\Delta = \emptyset$ is not a diagnosis candidate for the given observation. $\Delta = \{X_1\}$ is a diagnosis candidate for this instance, as assuming that X_1 is abnormal while every other component is normal makes the theory consistent. Indeed, since X_1 is assumed abnormal, we can no longer predict the value $out(X_1)$; and since we don't know the value $out(X_1)$ which is an input to both A_2 and X_2, we cannot predict neither $out(X_2)$ nor $out(A_2)$. Moreover, $out(A_2)$ is an input to O_1, thus without a value for $out(A_2)$ we cannot predict $out(O_1)$. Since we cannot predict $out(X_2)$ nor $out(O_1)$ anymore, we cannot have an inconsistency with the observed output $out(X_2) = 0, out(O_1) = 1$. All 22 diagnosis candidates are listed in Table 4.2. ♢

Example 4.4 illustrates that a simple diagnosis problem can have many candidates. Moreover, Table 4.2 shows redundancy in the set of all diagnosis candidates. Indeed, whenever $\Delta \subseteq \Delta'$ for a diagnosis candidate Δ, then Δ' is a candidate as well.

Table 4.2: Diagnosis Candidates for Example 4.4

$\{X_1\}$	$\{X_1, X_2\}$	$\{X_1, A_1\}$
$\{X_1, A_2\}$	$\{X_1, O_1\}$	$\{X_1, X_2, A_1\}$
$\{X_1, X_2, A_2\}$	$\{X_1, X_2, O_1\}$	$\{X_1, A_1, A_2\}$
$\{X_1, A_1, O_1\}$	$\{X_1, A_2, O_1\}$	$\{X_1, X_2, A_1, A_2\}$
$\{X_1, X_2, A_1, A_2\}$	$\{X_1, X_2, A_1, O_1\}$	$\{X_1, A_1, A_2, O_1\}$
$\{X_1, X_2, A_1, A_2, O_1\}$	$\{X_2, O_1\}$	$\{X_2, O_1, A_1\}$
$\{X_2, O_1, A_2\}$	$\{X_2, A_1, A_2, O_1\}$	$\{X_2, A_2\}$
$\{X_2, A_1, A_2\}$		

This property is proven as Proposition 3.4 in [66]. In the worst case, there could be exponentially many candidates. To partially circumvent this problem, Reiter considers only set-minimal candidates. Table 4.3 shows all set-minimal candidates for Example 4.4

Table 4.3: Minimal Diagnosis Candidates for Example 4.4

$\{X_1\}$	$\{X_2, O_1\}$	$\{X_2, A_2\}$

4.1.5 Rule-Based Diagnosis

OS discovery cannot be easily represented using model-based diagnosis: what is the system and how can we define it in terms of components such as to have operating systems as diagnosis candidates. Hence, we adopt a rule-based approach. We have to figure out which reasoning mechanism to use (abductive or consistency-based). This will also determine the kind of knowledge in SD (explanatory or fault-descriptive) and the format of the rules used to encode that knowledge. Following [61], we start by providing the format of the rules used to encode each kind of knowledge in SD (explanatory and fault-descriptive).

Definition 4.6 (Rules for Explanatory Knowledge)
Each rule gives the possible causes (explanations) for a specific observation. Rules have the form:

$$\mathit{EX}(c_1) \vee \mathit{EX}(c_2) \vee \ldots \vee \mathit{EX}(c_n) \leftarrow \theta_1 \tag{4.1}$$

where $c_i \in \mathsf{CONST}$ *and* $\theta_1 \in \mathsf{OBS}$*. We call* θ_1 *the antecedent of the rule and* $\mathsf{EX}(c_1) \vee \mathsf{EX}(c_2) \vee \ldots \vee \mathsf{EX}(c_n)$ *the consequent (*$EX(c)$ *means* c *explains the observation). We can assume that there is at most one rule with* $\theta_i \in \mathsf{OBS}$ *as its antecedent.* ◯

Definition 4.7 (Rules for Fault-Descriptive Knowledge)
For each possible cause we list the effects (observations, symptoms) associated with that cause (i.e., we describe the behavior of the system under a given "fault"). In this approach, rules have the form:

$$\theta_1 \leftarrow \mathsf{EX}(c_1) \tag{4.2}$$

We call $\mathsf{EX}(c_1)$ *the antecedent of the rule and* θ_1 *the consequent. Of course, it is possible to have multiple rules with the same antecedent. Equivalently, we can represent the set of rules having the same antecedent, say* $\mathsf{EX}(c_1)$*, into a single rule that would look like:*

$$\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n \leftarrow \mathsf{EX}(c_1) \tag{4.3}$$

◯

Now we will compare the two kinds of knowledge, based on the intuitive meaning of the rules and how each can handle incomplete knowledge. This will allow us to select the more appropriate kind of knowledge for OS discovery.

4.1.5.1 Intuitive Meaning of the Rules

Rule 4.1 intuitively means: "$c_1, c_2, \ldots, c_n$ are all possible individual explanations for observation θ_1". In other words, θ_1 is caused by at least one of $c_1, c_2, \ldots, c_n$.

An intuitive meaning of the rule 4.3 could be: "whenever c_1 is responsible for the behavior of the system, then observations $\theta_1, \theta_2, \ldots \theta_n$ will all occur". In other words, c_1 causes all of $\theta_1, \theta_2, \ldots \theta_n$ to occur. Unfortunately, this is not the intended meaning of such a rule (e.g., a disease does not automatically cause all its associated symptoms). The intended meaning is more along the lines of: "if c_1 is responsible for the behavior of the system, then observations $\theta_1, \theta_2, \ldots \theta_n$ *might* occur" (e.g., a disease *might* cause some of its associated symptoms). In other words, c_1 *might* be causing $\theta_1, \theta_2, \ldots \theta_n$. This intended meaning is not very intuitive because the notion of

"might occur" is not usually captured in classical logic, and definitely not understood as the meaning of a logical implication ($\leftarrow$). To circumvent this semantics problem, abductive reasoning is used.

4.1.5.2 Handling Incomplete Knowledge

We consider three situations where we can have incomplete knowledge:

Unanticipated Explanatory Constituent: CONST does not consider every possible explanatory constituent of the actual system. For instance, the disease flu is not included in the model.

Unanticipated Causal Relation: the model includes the explanatory constituent c and the observation θ, but does not include the fact that c might be an explanation for θ. For instance, we may fail to represent that chicken pox causes nausea.

Unanticipated Observation: OBS does not include every observation we can make about the system. For instance, the fact that we can observe the sex of the patient is not considered by the model.

Note that it is quite natural to have an incomplete diagnosis representation of a problem; some problems are simply too complex to model perfectly (e.g., medical diagnosis). Thus it is important to consider handling knowledge incompleteness. Below, we see how the two kinds of knowledge deal with these situations of incomplete knowledge.

4.1.5.2.1 Unanticipated Explanatory Constituent There is a cause c which can be an explanation for some observations, but we do not know about c (i.e., $c \notin \mathsf{CONST}$). In this case, both approaches will handle the incomplete knowledge in the same way: whenever c is part of the actual diagnosis, we will get an incorrect diagnosis, i.e, the actual diagnosis will not be part of the generated set of candidates. In the best case (this will mostly occur in the single fault setting), we will end up with an empty set of diagnosis candidates; i.e., we are unable to diagnose the system with

the available explanatory constituents and system description[1]. In general, we will end up with diagnosis candidates of higher cardinality to compensate (by blaming additional constituents) for the observations caused by c.

4.1.5.2.2 Unanticipated Causal Relation We know about cause c (i.e., $c \in \mathsf{CONST}$) and observation θ (i.e., $\theta \in \mathsf{OBS}$), but we are not aware that c can be an explanation for observation θ. Again here, both approaches behave in the same way. This will sometimes prevent us from considering c as being part of a candidate when it should be. Again this can lead us to be unable to diagnose the system or to generate candidates of higher cardinality (blaming extra constituents to compensate for c).

4.1.5.2.3 Unanticipated Observation There is an observation θ we could get from the system, but we do not know about it (i.e., $\theta \notin \mathsf{OBS}$). Here, there is a fundamental difference between the two approaches. In explanatory knowledge with consistency-based diagnosis, this means that there will be no rule with θ as its antecedent. Thus, θ has no discriminating power as it plays no role in the consistency of Definition 4.4.

Using fault-descriptive knowledge with abductive reasoning is more problematic. Here, θ would not appear in the consequent of any rule. From the definition of abductive reasoning (see Definition 4.5), a candidate must entail the observations. However, since θ is not the consequent of any rule, there is no way to derive θ. As a consequence, whenever we obtain an observation that was not anticipated, abductive reasoning is unable to compute any diagnosis candidate.

In our operating system discovery application, we expect plenty of unanticipated observations to occur. For that reason, we adopt the explanatory knowledge approach using consistency-based reasoning.

4.1.6 Diagnosis properties

To help us describe how operating system discovery can be seen as a diagnosis task, we introduce properties that define a specific diagnosis task (or help choose the proper

[1]Note that this is different from having $\emptyset$ as a candidate, in which case the system is not faulty (i.e., requires no explanation).

model to represent it). Some of these properties can already be handled by the diagnosis framework presented in section 4.1.1, while others require extensions (some of which have been proposed in the literature).

Fault Cardinality A system can be modeled as a single or multiple fault(s) system. In the single fault case, at most one explanatory constituent can be used to explain the observations. In the multiple faults case, multiple constituents can be responsible simultaneously. Fault cardinality can be handled by most diagnosis frameworks (see [66]).

Fault Behavior Faults can either be continuous or intermittent. Continuous faults are easier to diagnose because when we test the system we are guaranteed to get the faulty behavior.

System Knowledge What is the knowledge we have about the system: high-level knowledge of the components and their interactions (model-based diagnosis) or low-level knowledge of the symptoms and their causes (rule-based diagnosis)? Moreover, do we know how the constituents behave when faulty or when healthy, or both (this will dictate the reasoning mechanism to use for diagnosis)? This property was partially discussed in Section 4.1.3, and [26] proposes a general framework to handle both reasoning mechanisms simultaneously.

Stability The system can either be *static* or *dynamic*. A static system does not change by itself over time, while a dynamic one might[2]. A dynamic system can develop new problems (e.g., a patient may contract a new disease) while it is being diagnosed, or its configuration can change (e.g., a power distribution system can re-route power by itself, see [10]). Static systems are definitely easier to diagnose, and even if most systems are dynamic, we usually assume they are static when performing diagnosis. The ability to eliminate the hypotheses that do not explain the given observations works only with static systems.

Modifiability Some systems can be controlled by predefined actions. Performing

[2]Note that if a system is modified by an external agent other than the diagnosis tool, then the system can be considered dynamic.

some actions might be necessary before executing a diagnosis test (in order to put the system in the proper state).

Observability A system can be observed passively, actively, or both. In a passively observable system, one can only perform diagnosis with the observations provided by the system (e.g., during diagnosis, a factory chain can only be observed, and not acted upon to avoid interrupting the production). An actively observable system, on the other hand, only provides information as the result of predefined tests (e.g., an off-line electronic circuit will never produce inputs/outputs by itself). Finally, if a system supports both observation modes, then it can provide both active and passive information.

Reparability Can we fix the system while diagnosing it, or is the reparation part of the post-diagnosis process. Some interesting work has been done in diagnosing reparable systems, see [37] and [73]. However, reparable systems completely change the diagnosis approach. Even the objective changes from finding the explanation to restoring the systems back to a suitable state.

Prior Probabilities In some systems, even before having any observation, some explanations are more likely than others (e.g., some diseases are more common than others). In other systems, the prior probabilities of the explanations are all equal. Considerable work has been done on the use of prior probabilities of failure, mainly by de Kleer in [28, 29, 31].

Discrimination Power Sometimes, observations discriminate an explanation categorically (with 100% certainty). But in medical diagnosis, tests only modify the probability of the disease, a disease can rarely be eliminated with certainty.

Once the properties of a diagnosis problem have been established, building a tool for that specific problem sums up to implementing two algorithms: one to generate candidates based on observations, and another to eliminate candidates by probing the system for new observations. These two processes are introduced below.

4.2 Candidate Generation

Here we consider the work by Reiter concerning candidate generation [66]. We consider model-based diagnosis with consistency-based reasoning. This will be the building blocks for the implementation of our OS discovery tool.

The definition of a diagnosis candidate, Definition 4.4, appeals to a consistency test for arbitrary first-order formulae, which is undecidable in the general case. As a consequence, model-based diagnosis is undecidable in general. However, if we restrict SD to a decidable fragment of first-order logic, then diagnosis becomes decidable as well.

A naive algorithm, see Figure 4.3, to compute the diagnosis candidates would be to test the consistency of

$$\mathsf{SD} \cup \Theta \cup \{\mathsf{AB}(c) | c \in \Delta\} \cup \{\neg\mathsf{AB}(c) | c \in \mathsf{CONST} \setminus \Delta\}$$

for every $\Delta \in \mathcal{H}$, keeping those Δ for which the above theory is consistent as the diagnosis candidates. However, this procedure would be highly inefficient, as it would require $2^{|\mathsf{CONST}|}$ consistency checks.

NaiveCandidateGeneration(SD,CONST,Θ)

Provides the diagnosis candidates for Θ

Input: SD: the set of rules
CONST: the set of explanatory constituents
Θ: the set of observations to explain

Output: The set of diagnosis candidates

1 $\Gamma \leftarrow \emptyset$
2 FORALL $\Delta \in \mathcal{H}$
2.1 IF $\mathsf{SD} \cup \Theta \cup \{\mathsf{AB}(c) | c \in \Delta\} \cup \{\neg\mathsf{AB}(c) | c \in \mathsf{CONST} \setminus \Delta\}$ is consistent
2.1.1 $\Gamma \leftarrow \Gamma \cup \{\Delta\}$
3 RETURN Γ

Figure 4.3: Naive Algorithm to Compute Diagnosis Candidates in Multiple Faults

An easy improvement of the naive algorithm would be to consider the Δ in increasing order of cardinality and as soon as we find a diagnosis candidate Δ, we know

that all of its supersets are diagnosis candidates as well and these do not require consistency checks. However, in the worst case (i.e., if the only diagnosis candidate is $\Delta = \mathsf{CONST}$) we would still require an exponential number of consistency checks.

In [66], Reiter proposes another method of computing diagnosis candidates based on conflict sets and hitting sets. We provide the ideas of this algorithm here, because it will be the basis for our tool.

Definition 4.8 (Conflict Set)
A conflict set for a diagnosis problem instance is a set $\{c_1, \ldots, c_k\} \subseteq \mathit{CONST}$ *such that*

$$\mathit{SD} \cup \Theta \cup \{\neg \mathit{AB}(c_1), \ldots, \neg \mathit{AB}(c_k)\}$$

is inconsistent.

Δ is a diagnosis candidate iff $\mathsf{CONST} \setminus \Delta$ is not a conflict set.

Definition 4.9 (Hitting Set)
Suppose $\mathcal{C}$ *is a collection of sets. A hitting set for* $\mathcal{C}$ *is a set* $H \subseteq \cup_{S \in \mathcal{C}} S$ *such that* $H \cap S \neq \emptyset$ *for each* $S \in \mathcal{C}$*. A hitting set is minimal iff it has no proper subset that is also a hitting set.* ◯

Proposition 4.1 (Theorem 4.4 in [66])
Δ *is a (minimal) diagnosis candidate iff* Δ *is a (minimal) hitting set for the collection of conflict sets.*

Proof.
See proof of Theorem 4.4 in [66]. □

Proposition 4.1 suggests a new way of computing the set of minimal diagnosis candidates. First, compute the collection of minimal conflict sets. Then compute the minimal hitting sets for the collection of minimal conflict sets. Those minimal hitting sets are then the minimal diagnosis candidates.

Unfortunately, computing the set of minimal conflict sets seems to be as hard as finding the set of minimal diagnosis candidates. Moreover, finding a minimal hitting set is NP-Hard (as this problem is dual to finding a minimal set cover) [38].

The procedure discussed here contains three sources of intractability:

- the exponential size of $\mathcal{H}$.

- the consistency-check procedure.
- the hitting set procedure.

In [66] Section 4, Reiter proposes an algorithm to compute the minimal diagnosis based on a pruned hitting set data structure. However, this new algorithm does not have a better worst-case scenario than the procedure discussed above.

The concepts of conflict sets and hitting sets will be useful in our OSD tools. In Chapter 6, we'll see how the three sources of intractability can be addressed in the specific context of OS discovery. But for now, we need to introduce another important aspect of diagnosis: candidate elimination.

4.3 Candidate Elimination

Here we study the candidate elimination process of diagnosis. First, we discuss how tests are represented in order to allow reasoning (Section 4.3.1). Then, we present the current test selection strategy proposed in the diagnosis literature (Section 4.3.2).

4.3.1 Test Representation

In Section 4.1.1, we defined a diagnosis problem as a triple $\langle \mathsf{CONS}, \mathsf{SD}, \mathsf{OBS} \rangle$. Here we include a fourth component in the specification of a diagnosis problem: TEST. Tests are an important part of diagnosis, especially for candidate elimination [71]. In this section, we discuss the test representation used in this thesis, mainly focusing on how to reason with tests. But first, let us discuss a few properties related to the tests.

Execution Cost: An execution cost can be associated to each test. Examples of costs are: the money spent to perform the test, the time required to obtain the results, the inconveniences related to the execution of the test. In the simplest case, all tests have a cost of 0. Another simple case arises when all tests have the same cost. But in real life, different tests might have different costs.

Executability: Tests are not always executable. Some tests can only be executed when the system is in a specific state. Others can only be executed when specific

hypotheses have been ruled out.

Determinism: If we execute the same test twice in the exact same situation, will the results be identical?

For our application to OS discovery, we make the following assumptions on the properties of tests:

- All tests have the same execution cost. This is not necessarily true as we could see non-standard tests as being more costly than standard ones. Moreover, some tests simply generate more packets than others; these could also be considered more costly.

- Tests are always executable. Again, this is not always true as some tests require knowledge about one specific open/closed port on the target.

- Tests are deterministic. As far as we know, this is true in operating system discovery (i.e., sending the same stimulus twice will result in the same response).

Note that the first two assumptions are made for the sake of simplicity. They are not a consequence of a weakness in our test representation.

Below we briefly discuss two existing test representations, Reiter's [66] and McIlraith's [54] and finally present our *outcome-based* representation.

4.3.1.1 Reiter's Test Representation

Reiter briefly discusses candidate elimination in [66]. However, he uses *measurements* instead of tests. A measurement is simply a set of new observations. Reiter was not concerned about how to get a measurement, he focused on the impact of a measurement on the diagnosis candidates.

The main result of Reiter concerning the impact of a measurement is the following (see Proposition 5.3 in [66]). Given a hypothesis h and a measurement M, h *predicts* M iff:

$$\mathsf{SD} \cup \{\neg AB(c) | c \in \mathsf{CONST} \setminus h\} \cup \{AB(c) | c \in h\} \models M$$

where AB is a predicate representing the abnormal status of a constituent.

Based on this weak notion of prediction, we can see the impact of a measurement M on a set of candidates Γ. For every $h \in \Gamma$, h is not a candidate anymore based on M if h predicts $\neg M$.

This notion of prediction is weak in the sense that a given hypothesis may not predict any of M or $\neg M$, making the reasoning process difficult. Moreover, with non-boolean observations, the notion of a measurement is hard to relate to a test.

4.3.1.2 McIlraith's Test Representation

Most of the work around test representation for diagnosis has been conducted by McIlraith [7, 56]. In [54], McIlraith defines a test as a pair $\langle A, f \rangle$ where A is a conjunction of achievable literals (the precondition of the test) and f is the observable (a fluent). The two possible outcomes of that test are f and $\neg f$, i.e., after the execution of the test, either we know that f is true, or we know it is false.

A test $\langle A, f \rangle$ can be executed when $SD \wedge A \wedge h$ is satisfiable for every $\{h\} \in \Gamma$ (the set of current diagnosis candidates). The satisfiability of $SD \wedge A$ ensure that the preconditions A can be achieved (e.g., simultaneously making the input of a digital circuit to be 0 and 1 is not possible). However, the diagnosis candidates can further restrict the possible actions. For instance, the hypothesis that a patient is pregnant should prevent the execution of any X-ray test. For that reason, the formula must be satisfiable for every diagnosis candidate.

The limitation to two-valued fluents is cumbersome. It makes it hard to represent tests with several possible outcomes. Moreover, reasoning on the outcome of a test (and it's impact on the diagnosis candidates) is not intuitive (especially for tests with several outcomes). Below we provide a representation similar to this one, but directly oriented toward reasoning with tests.

4.3.1.3 Outcome-Based Test Representation

The outcome-based test representation [19] is inspired from McIlraith's representation discussed above, but is more suitable for OS discovery. It is particularly intuitive for reasoning about the possible effect of a test on the diagnosis candidates.

Definition 4.10 (Test)
A test t is a prediction function $P_t : \mathcal{H} \rightarrow \wp(\mathsf{OBS})$. That is, given a hypothesis $h \in \mathcal{H}$, $P_t(h)$ is the set of observations that we obtain when executing t in a situation where h is the actual diagnosis. ◯

Below is an example of a test in OS discovery.

Example 4.5 (OSD test)
One OSD test consists of sending a TCP SYN packet on a closed port of the target computer and analyzing the TCP RST/ACK packet it will produce as a response, this is Test-9 of Definition A.9. Its prediction function is partially given here:

- $P_{T9}(\text{Windows 2000 Sp1}) = \{tcp(yes, rstack, 255)\}$
- $P_{T9}(\text{MacOS 10.1.4}) = \{tcp(no, rstack, 64)\}$
- etc.

This basically means that if we send a SYN packet on a closed port of a computer running Windows 2000 sp1, then it will respond with a RST/ACK packet in which the DF bit is set and the TTL is 255. This is different from the behavior of MacOS, which would not set the DF bit and use a TTL of 64. ♢

Using the prediction function, we can define the possible outcomes of a test t.

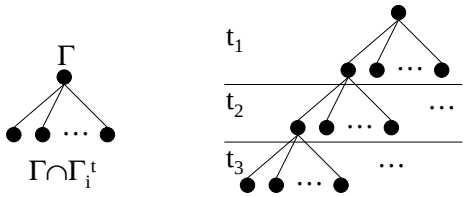
Definition 4.11 (Possible Outcomes of a Test)
Given a test t, the set of possible outcomes of t, denoted Θ_t is

$$\Theta_t = \{\theta | \theta \subseteq \mathsf{OBS} \text{ and } \exists h \in \mathcal{H} \text{ such that } P_t(h) = \theta\}$$

That is, $\theta \subseteq \mathsf{OBS}$ is a possible outcome of t if there is a hypothesis that will generate the observations θ in response to t. We denote $\{\theta_1^t, \theta_2^t, \ldots, \theta_k^t, \}$ the possible outcomes of test t. ◯

Each test outcome θ_i^t can be interpreted by a set of diagnosis candidates $\Gamma(\theta_i^t) \subseteq \mathcal{H}$ explaining the observations generated by this test outcome (see Definition 4.2).

Definition 4.12 (Interpreted Possible Outcomes)
A test t with possible outcomes $\{\theta_1^t, \theta_2^t, \ldots, \theta_k^t, \}$ is represented by its set of interpreted possible outcomes $\{\Gamma(\theta_1^t), \Gamma(\theta_2^t), \ldots, \Gamma(\theta_k^t)\}$. ◯



(a) Single Test (b) Test Sequence

Figure 4.4: Test Execution

Considering the execution of test t, in a situation where Γ is the current set of diagnosis candidates, leads to a tree of height 1 rooted at Γ with one leaf $\Gamma \cap \Gamma(\theta_i^t)$ for each possible outcome θ_i^t of t (see Figure 4.4(a)).

Considering the execution of the sequence of tests $S = [t_1, t_2, t_3]$, in a situation where Γ is the current set of diagnosis candidates, leads to a tree of height 3 rooted at Γ (see Figure 4.4(b)).

Note that the notion of a test presented here, i.e., based on a prediction function, has some limitations:

- The prediction function imposes the outcome of a test to be deterministic with respect to a specific hypothesis. That is, whenever h is the actual diagnosis, the result of executing test t will always be $P_t(h)$. This is not true in all diagnosis domains; in medicine, for instance, a given disease does not always produce the same test results; many factors must also be considered. In OSD, however, this should not be a problem as tests seem deterministic by nature.

- The prediction function requires that we know the outcome of the test for each element of $\mathcal{H}$. In the single fault case this is reasonable, but in the multiple faults case it becomes a problem as $\mathcal{H}$ grows quite large. Since we consider a single fault setting for OSD, this will not be a problem.

We can, and usually will, have cases with two hypotheses h, h' such that $P_t(h) = P_t(h')$. In that case, we say that h and h' behave identically with respect to t, or that t cannot distinguish between h and h'.

Example 4.6 (OSD test (continued from Example 4.5))
In Example 4.5, we considered Test-9 for OSD. We have seen that this test has at least two possible outcomes: $tcp(yes, rstack, 255)$ and $tcp(no, rstack, 64)$. We know that this test can distinguish between Windows 2000 sp1 and MacOS 10.1.4. However, other OSes can result in one of the two outcomes above. For instance, Windows XP Home sp2 and Windows 2003 server sp1 also provide the outcomes $tcp(yes, rstack, 255)$ for Test-9 (they are indistinguishable based on Test-9). On the other hand, most variants of FreeBSD, OpenBSD and NetBSD also provide the outcome $tcp(no, rstack, 64)$. ♢

By defining the outcomes of a test using observations, it is easier to connect candidate elimination with candidate generation in a diagnosis engine. More importantly, the notion of a prediction function makes the process of reasoning about tests very intuitive, as we directly know which outcome to expect under a given hypothesis.

Note that the definition of a test could be extended to include preconditions as well. We omit this extension here, for the sake of simplicity.

4.3.2 Current Test Selection Strategy

Current diagnosis tools focus entirely on the objective of finding the actual diagnosis. Consequently, the test selection strategies are all geared towards that goal. Surprisingly, there is only one test selection strategy discussed in the diagnosis literature: the greedy approach, which always executes the test that will refute the maximal number of diagnosis candidates. The consensus regarding this single test selection strategy is based on two claims:

- The computation time required for minimizing the number of executed tests is prohibitive compared to the cost of executing a few extra tests [31].
- The one step lookahead strategy employed by the greedy approach is "good enough" [30].

We disagree with these two claims.

For the first claim, we believe the tradeoff between computation time and test cost is domain dependent. When testing electrical circuits in a production environment,

the cost of a test is usually very low (sending a electrical signal as the input and measuring the output) while the time for testing each device might be limited (for productivity reasons). In that particular domain, executing extra tests might not be such a bad thing. In the medical domain, however, tests are extremely costly (in terms of time, money, and resources). In that context, a few minutes (even hours) of computation is nothing compared to the weeks required to obtain the test results. Thus, in the medical domain, minimizing the cost of testing is very important and there is plenty of computation time available.

For the second claim, we can rely on much stronger mathematical arguments to support or reject the claim. We can determine whether the greedy algorithm always returns the optimal solution or not. If not, we can determine whether it produces a bounded approximation of the optimal solution, i.e., regardless of the instance, does the greedy approach always return a solution whose size is a constant times the size of the optimal solution? This will be studied in Section 6.2.2.2.

4.4 Limitations

The theory of diagnosis provides a natural and elegant model for our OSD problem. An intrinsic memory (i.e., the diagnosis candidates), a safe process to manage/update this memory (i.e., candidate generation), and a way to select the proper tests to execute (i.e., candidate elimination) are the most important advantages of a diagnosis model for OSD. Chapter 6 will provide a more detailed explanation of our use of the theory of diagnosis to perform OS discovery.

Although the theory of diagnosis provides an interesting framework for OSD, it still has two limitations:

- By focusing entirely on the objective of finding the actual diagnosis, the theory does not adapt to the needs of the user.
- By considering only a single test selection strategy, the theory is not flexible enough to nicely adapt to the different priorities of different domains.

Before building an OSD tool based on the theory of diagnosis, Chapter 6 will first address these limitation by providing a general extension to the theory.

Chapter 5

Problem Statement

The problem is not the problem; the problem is our attitude about the problem.
-Dianna Booher

5.1 Objectives

In this thesis, we propose to build a new operating system discovery tool based on a hybrid approach. We have three main objectives regarding the development of such a tool:

- We want our tool to be *better* than every other existing tool.
- We want our tool to have a strong theoretical background.
- We want to provide a systematic (and automated) way of collecting OS fingerprints and incorporating them in our tool.

More information about these objectives and their importance is provided below.

5.1.1 A Better Tool

We want to build a *better* tool and we include several aspects in the notion of better.

- We want our tool to be more accurate than other tools (especially for the task of IDS context gathering).
- We also want our tool to be less intrusive than current active tools, i.e, send fewer network packets and as few malformed packets as possible (ideally none most of the time).

- We want our tool to be more flexible from the user point of view. The user should be able to ask queries in order to extract the information he actually needs.

We believe these improvements over existing tools will make our tool very attractive and usable by researchers, network administrators, and security officers. Our implementation should also be useable by a third party tool, something that is quite difficult to do with current OSD tools due to their ad hoc and informal flavor.

5.1.2 A Strong Theoretical Background

Current OSD tools are extremely ad hoc and do not rely on any theoretical background. We want to take a different approach because we believe that backing our tool against strong theoretical bases will have several benefits:

- It will give us insights regarding the computational complexity of our problem.
- It will provide us with well-established algorithms to implement the different modules.
- Improvements in the general theory can directly be applied to enhance our tool.
- It will serve as an elegant description of the tool engine.
- It will help identify and understand the assumptions/limitations of our tool.

5.1.3 Systematic Collection of OS Fingerprints

One of the problems underlying the OS discovery field is the gathering of OS fingerprints. Fingerprints are used to distinguish OSes in different situations. Usually, they take the form of rules (as seen in Chapter 2). Since several new OS versions are released each year, OSD tools must constantly be updated with the new fingerprints. Obtaining those new fingerprints is problematic. Current tools are doing that in an ad hoc way. Most tools rely on users submitting fingerprints through their website. However, this makes it difficult to control the quality of the fingerprints and to ensure a maximal coverage of all tests.

What we propose is a controlled virtual environment allowing us to gather the fingerprints in a systematic and automated way. Thus, it will be easy to obtain fingerprints for new OSes and since the experiments can be reproduced, we can validate the fingerprints.

5.2 Relevance

There are three main reasons why the work presented in this thesis is important. First, as pointed out in Section 3.1, information about the OSes is useful for several different tasks: from management to security related tasks. Second, as shown in Section 3.2, current OS discovery tools are not accurate enough to rely on their output, especially in the case of automated decision making. Finally, current OSD tools are not flexible enough; it is not possible for the user to directly extract the knowledge he is interested in. Instead, knowledge must be inferred from the output of a tool, usually given as human parsable string[1].

5.3 Methodology

To achieve our objectives, we proceed in three steps detailed in Chapter 6:

- First, we develop a hybrid approach to OS discovery based on the theory of diagnosis. Our approach combines the advantages of the classical ones (active and passive) while avoiding their limitations. Moreover, the hybrid approach is knowledge-oriented, making it more flexible from the user point of view. The final product, as supported by the experimental results in Chapter 7, is an OSD tool which is more accurate than current state-of-the-art tools.

- Then, we generalize the theory of diagnosis with a query-based extension. We also adapt some algorithms for the specific needs of OSD.

- Finally, we develop a virtual environment to automatically execute network experiments. This environment allows us to gather OS fingerprints quickly and cheaply.

[1] OSD tools sometimes provide unformatted outputs such as: “Windows 2000 sp2-sp4”, “Windows 2000 SP2+, XP SP1 (seldom 98)”, or “FreeBSD 4.8-5.1 (or MacOS X 10.2-10.3)”

5.4 Evaluation and Validation

We will measure the success of our OSD tool using the same experiment presented in Chapter 3. This will allow us to compare our implementation with other OSD tools, see Chapter 7.

Below we discuss the requirements for an ideal evaluation of OSD tools. We also argue why we are confident in our evaluation, even though it does not meet all the requirements for an ideal evaluation.

5.4.1 Ideal OSD Evaluation

We identify five guidelines for ideal OS discovery evaluation.

- Use different network topologies. The network topology can affect OSD tools in two different ways: traffic visibility and traffic modification. Some traffic may not be seen by the OSD tool depending on the network topology. For instance, if the tool and the fingerprinted target are not on the same network segment, then the ARP request from the target will not be seen by the tool. Moreover, if a packet needs to go through a router before reaching the OSD tool, then it might be modified on its way. For instance, routers will decrease the TTL value of IP packets.

- Provide OS diversity. If all the computers in the dataset used for evaluation are running the same OS (say Linux RedHat 9.0), then an OSD tool could be artificially good by always guessing Linux RedHat 9.0. That tool, however, would perform very poorly when evaluated on other datasets.

- Provide application diversity. Two different web servers might force different packet configurations, e.g., TCP options. As a result, an OSD tool relying heavily on TCP options to identify OSes could make mistakes when facing two Linux machines with different web servers.

- Provide traffic diversity. If the traffic traces represent only large file downloads, then chances are we will only see some specific TCP traffic. Thus, passive tools relying on ICMP traffic will perform artificially poorly.

- Performing several evaluations on different and independent datasets. This increases confidence in the results and generalizes the conclusions. A tool being the best on a single dataset is not as significant as a tool being the best on all of ten datasets. Unfortunately, it is extremely difficult to have access to even one such dataset.

5.4.2 The Evaluation Dataset

With respect to the guidelines above, the dataset we are using (see Appendix B) has the following properties:

- Limited network topology. The dataset uses a simple topology where all the computers are on the same network segment. However, since most OSD tools handle TTL fluctuation, this should not be a huge concern.

- Very good OS diversity. By using 95 different OSes from several families, the dataset allows us to really evaluate how good are OSD tools. Unfortunately, some important OS families, such as MacOS and SunOS, are not represented in the dataset.

- Very good application diversity. By using many different applications and several versions of the same application, we maximize the chances to have one OS generate slightly different traffic.

- Good traffic diversity. Since the dataset relies on the execution of exploits, the resulting traffic is quite diverse. However, the traffic does not reflect what is usually seen on a corporate network.

- Only one dataset. Unfortunately, it is very difficult to obtain a dataset for OSD evaluation; we need both traffic traces (often causing privacy issues) as well as a direct access to the computer used to generate them. We are currently not aware of an environment other than the one we used (*vlab* from CRC) which would be adequate for our experiment.

Although our experiment does not meet all the criteria for an ideal evaluation of OSD tools, we are confident that the results are representative; mainly because of the

high diversity of the dataset. Moreover, to our knowledge, this is the most thorough experimental evaluation of OSD tools in the literature.

Chapter 6

Towards Better Operating System Discovery

Success is not the destination, its the journey.
-Arthur Robert Ashe, Jr.

This chapter presents the three main contributions of this thesis. First, Section 6.1 introduces our new hybrid approach to OS discovery. Section 6.2 extends the theory of diagnosis to address the particular needs of OS discovery. Finally, Section 6.3 presents our virtual network environment that can be used to automatically collect OS fingerprints.

6.1 A Hybrid Approach to Operating System Discovery

In Section 3.2, we saw that current OSD tools do not provide very good results in practice. By studying the classic active and passive approaches, we established their main limitations: the absence of memory and the lack of reasoning capabilities.

This section presents hosd, our hybrid approach to operating system discovery that encompasses the advantages of the two classic approaches while being more flexible. Among other things, our approach is less intrusive than current active tools, because it carefully selects which active tests to execute, and uses passively gathered information to eliminate some possibilities. It is also more accurate than any classic OSD tool, as it has a memory and can rely on active tests when needed.

Moreover, we will see how the active and passive modules interact together to fulfill the user's needs. More information about the actual implementation of our hybrid approach will be provided in Chapter 7.

6.1.1 The General Picture of Hybrid OS Discovery

In hybrid OS discovery, the tool continuously monitors the network to passively gather as much information as possible. It goes into active mode only when needed (i.e., when a query is made that cannot be answered with the available knowledge) and then uses the information gathered passively to minimize the number of active tests performed. Example 6.1 shows a situation when a hybrid tool goes into active mode.

Example 6.1 (hybrid OS discovery)
Suppose a user wants to know if machine I is running Windows 2000 Server sp1, but the information gathered passively only allows us to deduce that it is running a Windows OS. Here we will use active tests to answer the query. However, only tests that discriminate between different Windows OSes will be considered (other tests are irrelevant here). For instance, there would be no point in executing a test that distinguishes between Linux and Windows, as we already know that I is not running Linux. ♢

The hybrid approach offers several advantages over the active and passive ones, see Table 6.1. First, constantly monitoring the network implies using a knowledge management system to implement a memory. This offers the possibility of ensuring the convergence of the set of possible OSes and to detect (and react to) certain network events (e.g., reboot, IP change). Second, the objective of only executing tests that are necessary implies the use of test selection strategies. This is an opportunity to provide more flexibility as to the kind of queries the system can answer, i.e., not only "What is the actual OS running?" but also "Is the computer running the given OS o?" as well as "Is the computer running an OS in the given set O?"; and to restrict the generation of malformed traffic. Finally, combining the active and passive approaches will reduce the amount of traffic generated (and also possibly the amount of time required) for OS discovery purposes (compared to active tools) while achieving the required level of accuracy (which is not always the case with passive tools). For instance, port scans can often be avoided by using passively gathered knowledge to find an open and/or a closed port.

Table 6.1: Advantages/Inconvenients of OSD Approaches/Tools

OSD Tools	Advantages	Limitations
Passive	• Access to purely passive tests • Non-intrusive	• No access to purely active tests • No access to information on demand • No memory • No continuous monitoring • Single-packet analysis • Ad hoc and informal
Active	• Access to information on demand • Access to purely active tests	• Oriented toward a single query • No continuous monitoring • Intrusive • Ad hoc and informal
Hybrid	• Access to purely active tests • Access to purely passive tests • Access to information on demand • Limited intrusion • Continuous monitoring • Supports multiple queries • Multi-packets analysis • Memory • Formal and based on theoretical grounds	

6.1.2 The Passive Module

The passive module has to fulfill two important requirements:

- Continuously monitor the network.
- Maintain a set of possible OSes that converges towards the actual operating system (i.e., the actual OS should always belong to the set of possible OSes).

To achieve this, we consider the passive module as a simple knowledge management module. The dynamic knowledge base will consist of the set of currently possible OSes for a given computer. At the beginning, the knowledge base is initialized so that every existing OS is possible. When new information is obtained, OSes that are guaranteed not to be the actual OS are removed from the set of possible OSes. So the idea of the passive module is to use the packets to eliminate OSes that cannot generate them.

This knowledge-based method differs from the guess-based method of current OSD tools (As seen in Chapter 2). The most important difference is the way we can

interpret the current set of possible OSes in order to answer a query.

6.1.2.1 Querying the Knowledge Base

As mentioned in Section 3.1, OS discovery is useful in various situations that can be grouped under three queries: Single OS Query, Group OS Query, and Exact OS Query.

Managing a knowledge base (i.e., set of possible OSes) makes it quite easy to answer those queries. Let P be the current set of possible OSes:

Single OS Query for o: if $o \notin P$, then the answer is *no*. If $P = \{o\}$, then the answer is *yes*. Otherwise, the answer is *unknown*.

Group OS Query for O: If $O \cap P = \emptyset$, then the answer is *no*. If $P \subseteq O$, then the answer is *yes*. Otherwise, the answer is *unknown*.

Exact OS Query: if $P = \{o\}$, then the answer is o. Otherwise, the answer is *unknown*.

Note that with a guess-based approach (as it is the case with classical OSD tools), the answer to all of these queries is always unknown. The reason is simple: at any time a new OS might appear in the set of possible OSes and that could change the answer to the query.

Sometimes it will be possible to answer a query simply based on the knowledge gathered passively. In other cases, when the answer is unknown, we need more knowledge; this is exactly when the active module comes into play.

6.1.3 The Active Module

The active module of an OSD tool sends specific stimuli (tests) to a target and gathers the target responses as new information. In typical active OSD tools, the information from several tests is correlated to provide a final answer. In our hybrid approach, we use the information from a single test to remove some OSes from the set of possible OSes.

The main drawback of active OSD tools is that all tests have to be executed every time a query is made, since the information from all tests has to be correlated.

In our hybrid tool, we want to limit the number of active tests used (to limit the amount of traffic generated). We do this in three closely related ways:

- By relying heavily on the information gathered passively: in the ideal case, the current knowledge-base is sufficient for answering the query, thus we don't have to execute any test.

- By using a query-based approach: the idea is that the Single OS Query generally requires fewer tests than the Exact OS Query[1], see Section 6.2.1.3. For instance, if the user only wants to know if the computer is running Windows 2000 sp2, it is not always necessary to learn the exact OS. Sometimes, one test will be sufficient to learn that it is not running Windows 2000 sp2 (i.e., to remove it from the set of possible OSes).

- By reasoning, to select judicious tests: instead of executing all tests, like active tools do, we only perform *relevant tests* (those tests that will/might eliminate some OSes from the possible OSes set) and perform the most relevant ones first. Moreover, using reasoning in order to select tests allows us to avoid, or limit, the injection of malformed packets (which is a known drawback for some active tools).

Based on the description of the hybrid OS discovery task given so far, Section 6.1.4 will illustrate how it can be represented as a diagnosis problem.

6.1.4 Hybrid OSD as a Diagnosis Task

We represent the OSD diagnosis task as a quadruple $\langle \mathsf{CONST}, \mathsf{OBS}, \mathsf{SD}, \mathsf{TEST} \rangle$. Each element is discussed below.

[1]Current active OSD tools always try to find the exact OS

6.1.4.1 Explanatory Constituents (CONST)

In operating system discovery, CONST is the set of operating systems considered. By analogy with medical diagnosis, we will say that the "disease" (resp. operating system) of a specific "patient" (resp. computer) is, for instance, chicken pox (resp. *Windows* XP sp3). Moreover, we consider only single fault diagnosis. As a consequence, our hypothesis space, from which we select possible diagnoses, is $\mathcal{H} = \{\{c\}|c \in \mathsf{CONST}\} \cup \{\emptyset\}$. A diagnosis candidate Δ is thus a single element of CONST, i.e., a single OS. We can interpret $\Delta = \{c\}$ as the conjecture that the computer is running the OS represented by c.

There is one case where it would be interesting to consider multiple faults diagnosis for OSD: when multiple computers are hidden behind a network address translator (NAT[2]). In such a case, the traffic coming from those machines will appear to come from the NAT, but it will represent different OS behaviors (for the different hidden computers). This NAT situation will pose a problem to our single fault model (no hypothesis can explain the observations generated by a NAT), but could be addressed nicely with a multiple faults model. The single fault model is adopted for the sake of simplicity. The multiple faults model is left for future work.

6.1.4.2 Observations (OBS)

Observations in OSD are network events. For simplicity's sake, we usually consider an observation to correspond to a single packet. But, it could also be a more abstract network event such as 3 ARP requests with a delay of 6 seconds in between, or a stimulus-response pair of packets (e.g., TCP SYN and TCP SYN/ACK). In practice, a network event never contains more than a few packets. The observation space is quite large, and we don't expect our diagnosis system to know about all of it. Most network packets are irrelevant from an OSD point of view.

6.1.4.3 System Description (SD)

We use a rule-based approach for OSD. This is quite natural, since we could hardly imagine a model of the underlying system. It is not even clear what is that system in

[2]Either a physical NAT device or a virtual NAT architecture.

terms of constituents. One of the main criticisms of rule-based diagnosis is its close relationship to expert systems, in which the rules are provided by experts in a very ad hoc manner. However, one of our three objectives, as stated in Chapter 5, is to provide a fully automated way of gathering OS fingerprints and incorporating them into our tool. This will result in a rigorous and automatic way of generating SD. This will be discussed in Section 6.3.

The rules composing SD will have the form

$$c_1 \vee c_2 \vee \ldots \vee c_n \leftarrow \theta \tag{6.1}$$

where $c_i \in \mathsf{CONST}$ and $\theta \in OBS$. That is, for each observation (network event), we have the complete set of possible causes (operating systems) that can explain it. Those rules represent what we called explanatory knowledge in Section 4.1.5. We could have used the rule format advocated by [61] ($\theta \leftarrow c$), encoding fault-descriptive knowledge; however, we believe rules like (6.1) to be better suited for our task. We based our decision on the discussion we had of Section 4.1.5 about the two rule formats. Three arguments mainly support our decision:

- Explanatory rules are more intuitive.
- Explanatory rules handle unanticipated observations without any problems, while fault-descriptive rules don't. We expect to come across many unanticipated observations in OSD.
- Explanatory rules lead to consistency-based reasoning. From there, and using Reiter's work described in Section 4.2, we can derive a fast algorithm for candidate generation in OSD. This algorithm will be presented in Section 6.1.4.6.

Based on the content of SD and the discussion in [63], we use the consistency-based definition of a diagnosis candidate for OSD.

6.1.4.4 Tests (TEST)

The tests are OS discovery tests as discussed in Section 2.2.1. They are represented in the outcome-based format as presented in Section 4.3.1.3.

6.1.4.5 Properties of the OSD Diagnosis Task

To have a better understanding of the OSD diagnosis task and the assumptions we rely on, we consider the diagnosis properties introduced in Section 4.1.6 from the OSD point of view.

Fault Cardinality We assume OSD to be a single fault diagnosis problem. However, some situations, like NATed computers, would require a multiple faults model.

Fault Behavior Faults are continuous. A computer will always exhibit the behavior of its underlying OS.

System Knowledge We are using low-level knowledge (rules) of the system together with explanatory rules.

Stability We assume the system to be completely static, i.e., a computer will never change its OS. This assumption is a simplification for two reasons. First, a user can install a new OS on his machine, thus violating the static assumption. Second, we associate an OS to an IP address (and not really to a computer). The binding IP-OS is not entirely static (e.g., DHCP renew, address conflicts, etc.). Thus, OSD is not entirely static, but it is static enough for this assumption to hold most of the time. We could deal with the dynamic nature of OS discovery by identifying and monitoring the network events breaking the IP-OS association (reboot, DHCP requests). Upon such an event, the set of possible OSes needs to be reset (all OSes are now possible). This interesting extension is left for future work.

Modifiability The system cannot be modified by the diagnosis tool through actions. Indeed, an OS discovery tool cannot change the operating system of a machine, nor the network topology.

Observability OSD is both passively and actively observable. Some events can only be observed passively, while some others are only observed after a specific stimulus. Moreover, due to the network topology, some events might not be observable (e.g., ARP packets do not travel outside a network segment).

Reparability The notion of reparability does not apply to OSD, since nothing is broken. OSD is really diagnosis in the sense of explaining the system's behavior and not finding out what is wrong.

Prior Probabilities We assume all hypotheses to be initially equiprobable. However, some OSes (Windows and Linux) are clearly more likely to run than others (Pico BSD and BeOS). Thus, considering non equal initial probability distribution seems an interesting avenue for OSD. It is hard, however, to obtain reliable individual prior probabilities, i.e., for each OS and not for a whole family. Moreover, those probabilities may vary significantly for different networks, e.g., we can expect the OS distribution from Microsoft headquarters to be significantly different than the one from a Sun development lab. Using probabilities requires to adapt the candidate generation algorithm (to update the probability of each candidate). It could be used to guide the test selection strategy (e.g., to select the "best" ordering of tests). This interesting avenue is left for future work.

Discrimination Power We consider that observations discriminate categorically. For instance, if we see a packet from a computer and the packet cannot be generated by a Windows system, then we conclude that the computer is not running Windows.

Table 6.2 summarizes the diagnosis properties of OSD and emphasizes the assumptions we are making when modeling OSD for the implementation of our hybrid approach.

6.1.4.6 Candidate Generation Algorithm for OSD

Based on the candidate generation algorithm provided by Reiter in [66] (also discussed in Section 4.2) and on the properties of the OSD diagnosis task, we design a fast algorithm for candidate generation. This algorithm can be used for any diagnosis problem having the same properties as OSD.

Our algorithm, see Figure 6.1, first computes the minimal conflict sets for a given collection of observations and then computes the minimal hitting sets of those conflict sets, thus providing the minimal diagnosis. Considering the fact that OSD is modeled

Table 6.2: Diagnosis Properties for OS Discovery

	Actual OS Discovery	**Our Model**
Fault Cardinality	Multiple Faults (rarely)	Single Fault
Fault Behavior	Continuous	Continuous
System Knowledge	N/A	Rule-Based & Consistency-Based
Stability	Weakly Dynamic	Static
Modifiability	No	No
Observability	Passive & Active	Passive & Active
Reparability	N/A	N/A
Prior Probabilities	Different	Equal
Discrimination Power	Conjectured Complete	Complete

as single fault diagnosis and using the structure of SD, we explain how these two tasks can be executed in polynomial time.

When analyzing the algorithms, we consider the following assumption about SD.

Assumption 6.1

Two distinct rules in SD have distinct antecedents (observations).

We can enforce this assumption when creating SD by merging the rules with the same antecedent into a single one.

6.1.4.6.1 Algorithm Analysis Parameters Before we provide the two algorithms and their analysis, let us consider what are the parameters for the analysis:

- |SD| Currently, we have 433 rules, thus |SD| = 433. This number is constant from one run to the next. It only changes when we regenerate SD (to include new OSes or new OSD tests).

- |CONST| Currently, we consider 208 OSes, so |CONST| = 208. Once again, this number is mainly constant. It only changes when we integrate a new OS. We consider two versions of the same OS to be two distinct OSes, e.g., Windows 2000 sp1 and Windows 2000 sp2.

GeneralCandidateGeneration(SD,CONST,Γ,Θ)

Provides the diagnosis candidates for Θ

Input: SD: the set of rules
CONST: the set of explanatory constituents
Γ: the current set of diagnosis candidates, initially $\mathcal{H}$
Θ: the set of observations to explain

Output: The set of diagnosis candidates

1	$\mathcal{C} \leftarrow$ **ConflictSetsGeneration**(SD,Θ)	$O(\|\mathsf{SD}\| \times \|\Theta\|)$
2	$H \leftarrow$ **HittingSetsGeneration**($\mathcal{C}$)	$O(\|\mathsf{CONST}\| \times \|\Theta\|)$
3	RETURN $\Gamma \cap H$	

Figure 6.1: General Candidates Generation Algorithm

- $|\Theta|$ (the set of given observations). In our implementation, we will keep the size of Θ below a certain threshold (currently fixed at 100). After a run of candidate generation, the observations are discarded; only the diagnosis information they convey is kept by remembering the resulting set of diagnosis candidates (and using it for the next run of candidate generation).

6.1.4.6.2 Conflict Sets Generation First, we consider an alternative notation for the rules in SD. We transform each rule r of SD into r' in the following way. Given $r \in \mathsf{SD}$

$$r := EX(c_1) \vee \ldots \vee EX(c_{n_r}) \leftarrow \theta_r$$

the resulting rule r' is

$$r' := set_r \leftarrow \theta_r$$

where $set_r = \{c_1, \ldots, c_{n_r}\}$. Thus, the consequent of the rule is transformed into a set representing the disjunction of possible explanations for the observation. The rule r' is now interpreted in the following way: if we observe θ_r, then the possible explanations are set_r.

Now we extract an important property of the new rule format.

Property 6.1

Given the observation θ_r and the rule r' presented above, the conflict set is exactly set_r.

ConflictSetsGeneration(SD,Θ)

Provides the conflict sets for Θ

Input: SD: the set of rules sorted
Θ: the set of observations to explain

Output: The set of conflict sets

```
1    C ← ∅
2    FORALL r ∈ SD
2.1      FORALL θ ∈ Θ
2.1.1        IF θ matches r
2.1.1.1          C ← C ∪ {set_r}
3    RETURN C
```

Figure 6.2: Conflict Sets Generation Algorithm

Based on the above property, we present a polynomial time algorithm to compute the conflict sets, see Figure 6.2.

The algorithm of Figure 6.2 runs in $O(|\mathsf{SD}| \times |\Theta|)$, assuming the union operation performed in step 2.1.1.1 takes unit time[3].

Note that we do not believe $O(|\mathsf{SD}| \times |\Theta|)$ to be a tight analysis. We can probably achieve faster results by sorting SD (off-line) according to their antecedent and sorting Θ (on-line). This would allow us to avoid going $|\mathsf{SD}|$ times through the elements of Θ.

6.1.4.6.3 Hitting Sets Generation For the algorithm computing the hitting sets, we have to consider our special situation: we are looking for single fault candidates only. In other words, we are looking for diagnosis candidates of cardinality one. According to Proposition 4.1, we know that each candidate is a hitting set of the conflict sets. Thus, we are interested in singleton hitting sets. This leads us to the following proposition:

Proposition 6.1

$\{h\}$ is a singleton hitting set of the collection $\mathcal{C}$ iff $h \in \cap_{S \in \mathcal{C}} S$.

[3]This can be achieved by using a symbolic wrapping of the sets. Then, before returning, we unwrap the sets. The unwrapping of a set can be done in constant time using indexed random access storage.

HittingSetsGeneration($\mathcal{C}$)

Provides the singleton hitting sets for $\mathcal{C}$
Input: $\mathcal{C}$: the collection of conflict sets
Output: The hitting sets

1 $H' \leftarrow \emptyset$
2 $H \leftarrow$ **firstElem**($\mathcal{C}$)
3 FORALL $S \in \mathcal{C}$
3.1 $H \leftarrow H \cap S$
4 FORALL $h \in H$
4.1 $H' \leftarrow H' \cup \{\{h\}\}$
5 RETURN H'

Figure 6.3: Hitting Sets Generation Algorithm

Proof.

$\Rightarrow$: *Assume* $\{h\}$ *is a singleton hitting set of the collection* $\mathcal{C}$. *By definition of a hitting set (see Definition 4.9),* $h \in S$ *for all* $S \in \mathcal{C}$. *Thus,* $h \in \cap_{S \in \mathcal{C}} S$.

$\Leftarrow$: *Assume* $h \in \cap_{S \in \mathcal{C}} S$. *Thus* $h \in S$ *for all* $S \in \mathcal{C}$, *which means, by definition of a hitting set (see Definition 4.9), that* $\{h\}$ *is a hitting set of* $\mathcal{C}$. *It is clearly a singleton.* □

The proposition above provides a direct algorithm to compute the singleton hitting sets (Figure 6.3).

This algorithm requires $O(|\Theta| \times |\mathsf{CONST}|)$. Since $H \subseteq |\mathsf{CONST}|$, steps 4 and 4.1 runs in $O(|\mathsf{CONST}|)$. Step 3 seems to be a problem because there are $2^{|\mathsf{CONST}|}$ distinct conflict sets. However, $\mathcal{C}$ cannot contain more than $|\mathsf{SD}|$ conflict sets (as each rule provides at most one) and $\mathcal{C}$ cannot contain more than Θ conflict sets as each observation appears in (i.e., triggers) at most one rule (see Assumption 6.1). Thus, if we consider that the intersection of 3.1 takes $O(|S|)$ which is no greater than $O(|\mathsf{CONST}|)$, then we can conclude that step 3.1 runs in $O(|\Theta| \times |\mathsf{CONST}|)$. Finally, the whole algorithm runs in $O(|\Theta| \times |\mathsf{CONST}|) + O(|\mathsf{CONST}|) = O(|\Theta| \times |\mathsf{CONST}|)$

The three sources of intractability discussed in Section 4.2 were handled in the following way:

- the size of $\mathcal{H}$ is not exponential in the single fault case.
- the consistency-check procedure is polynomial, thanks to the specific rule format of OSD.
- the hitting set procedure is polynomial when hitting sets are singletons (which is the case in the single fault case).

6.1.4.6.4 Impact The algorithm presented in Figure 6.1 runs in $O(|\mathsf{SD}| \times |\Theta|) + O(|\mathsf{CONST}| \times |\Theta|)$ which is polynomial. The impact of such a fast candidate generation algorithm is direct for our OSD problem: we can use this polynomial algorithm for the passive module of hosd.

6.2 Extending the Theory of Diagnosis

In the previous section, we saw how OS discovery is nicely modeled by using the theory of diagnosis. As mentioned in Section 4.4, the theory of diagnosis has two limitations which are inconvenient in the case of OS discovery: the user cannot query the knowledge base (other than by asking what is the actual diagnosis) and there is only one studied test selection strategy (the greedy one).

Below, Section 6.2.1 proposes a query-based extension to the general diagnosis model together with three queries that are in complete correspondance with the queries we expect an OSD tool to handle. Moreover, an argument that a query-based approach is both meaningful and useful in other domains than OSD is provided.

Then, Section 6.2.2 initiates a study of the test selection strategies related to each query. This study will again support the usefulness of the query-based approach and will also indicate the importance of considering other selection strategies than the greedy one.

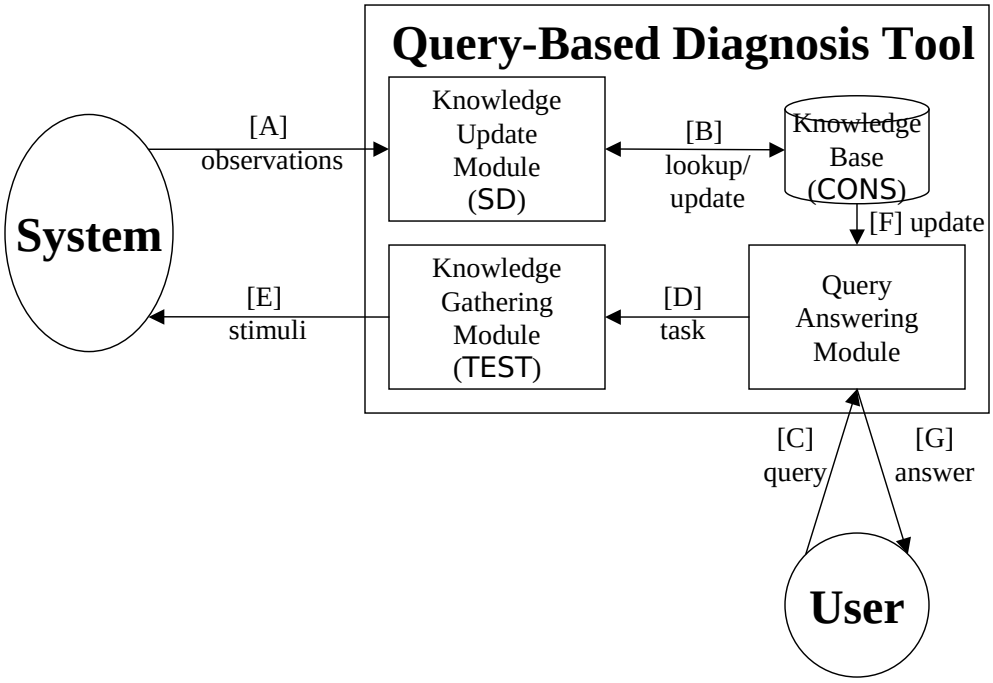


Figure 6.4: Query-Based Diagnosis Tool Behavior

6.2.1 Query-Based Extension

Figure 6.4 presents an extended architecture for diagnosis. It is very similar to Figure 4.1, but it contains the user interaction through queries. The operation mode is described below:

A: The diagnosis tool obtains observations from the system.

B: The diagnosis tool then computes the possible explanations for the observations and updates its knowledge base.

C: At some point, the user queries the tool. If the current set of candidates allows to answer the query, then we go to step G. Otherwise, we go to step D.

D: The diagnosis tool then selects a test that will generate new observations. The selected test should be the *best* (e.g., the most discriminant, the least costly, or the fastest) with respect to the specified query and the current state of the knowledge base.

E: The selected test is executed. This will stimulate the system and generate new observations (that will be processed in A and B above).

F: The updated knowledge will be considered and we go back to step C: answer the query if possible, select another test otherwise.

G: When the system has sufficient knowledge to answer the query, the user is notified and testing stops.

This extension provides much more flexibility to the user and it opens the doors to studying different test selection strategies. The idea is that different queries will allow different heuristics to guide the test selection strategy. This will be discussed in more detail in Section 6.2.2.

6.2.1.1 Diagnosis Queries

The extension we proposed above allows the user to query the diagnosis tool. To do so, we must define the queries that can be used. One straightforward query would be to get the current set of candidates. This query is easy because it never requires reasoning from the diagnosis tool. In this thesis, we consider the following three queries:

Single Candidate Query: given a hypothesis $h \in \mathcal{H}$, is h the actual diagnosis?

Group Candidate Query: given a set of hypotheses $H \subseteq \mathcal{H}$, is the actual diagnosis included in H?

Exact Candidate Query: what is the actual diagnosis?

The Exact Candidate Query is the only query considered by current diagnosis tools.

It is easy to see the relevance of these queries to the OS discovery domain. Simply notice the similarity between the above queries and the three OSD queries we defined in see Section 3.1. In fact, the Single Candidate Query corresponds to the Single OS Query, the Group Candidate Query to the Group OS Query, and the Exact Candidate Query to the Exact OS Query.

However, these queries are also meaningful in other diagnosis domains. This makes our query-based extension a general one.

6.2.1.2 Meaningfulness of Diagnosis Queries

To be convinced that the queries are meaningful, we consider them in two other diagnosis contexts: medical diagnosis and engineering diagnosis.

6.2.1.2.1 Medical Diagnosis Consider a medical domain where tests can take a very long time before the results are available. It is reasonable to think that in some cases when a patient comes to a doctor with serious symptoms, the first step would be to rule out the possibility of the patient being contagious with a dangerous disease. This would amount to asking the query "Does the disease belong to the set of contagious diseases?". If the patient is contagious, then he would immediately be placed in quarantine (before even knowing the actual disease). Otherwise, since the possibility of contagion has been ruled out, the diagnosis task can then proceed to the next step. The second step could be to rule out the need for a heart transplant (maybe because the process of a heart transplant is extremely long and must be started as soon as possible to maximize the chances of success). The query could be "Does the disease belong to the set of diseases requiring a heart transplant?".

Again in the medical domain, assume a patient, who has been successfully treated for Myeloid Leukemia in the past, goes to a doctor with strange symptoms. One of the main concerns should be to test for a possible relapse. In that situation, the priority is to answer the query "Is Myeloid Leukemia the actual disease?". If the answer to the query turns out to be negative, then the diagnosis process can continue normally. Otherwise, treatments should be resumed immediately.

6.2.1.2.2 Engineering Domain In other diagnosis domains, such as testing of physical devices (e.g. circuits), the new queries are also meaningful. For the sake of our examples, we consider components as entities that can fail and parts as entities that can be replaced. When working with a physical device, it is quite common that parts are sets of components. So even if two components can fail independently (one can fail while the other still functions properly), they will both be replaced simultaneously if they belong to the same part. When this is the case, it is quite irrelevant to know exactly which component has failed, we are more interested to

know which part has to be replaced. For instance, assume we have a device with five components $c_1, \ldots c_5$ and two parts p_1, p_2. Part p_1 contains components c_1, c_2, and c_3 while p_2 contains the other two components. If the device is not working properly, we are not so much interested in knowing which component is broken as we are to know which part should be replaced. So, instead of asking "Which component is broken?", we could ask "Does the broken component belong to p_1 (i.e. $\{c_1, c_2, c_3\}$)?". If it does belong to p_1, then p_1 needs to be replaced, otherwise p_2 needs to be replaced.

In a similar way as in the medical example, we could have a high priority component c (maybe it is extremely expensive, maybe it takes a very long time to replace) such that if the device is not working properly, we are mostly interested in knowing if c is broken or not. The query "Is c the broken component?" would be adequate for that situation. Similarly, there are many household devices that will get fixed only when a single given component is defective. A cheap clock (or a flashlight, or a watch) usually gets fixed only when the batteries are dead. If the batteries are good, it is quite rare that someone will disassemble the clock to fix the mechanism. In that case, we are not interested in knowing why the clock is not working, we are solely interested in knowing if the batteries are dead. So the query "Are the batteries dead?" (or more generally "Is Δ the actual diagnosis?" where Δ represents the batteries) is used here.

These examples are just a few among those where the new queries would be meaningful.

6.2.1.2.3 Discussion The examples above, both from the medical and engineering domains, show that the queries are meaningful in different diagnosis domains. However, this conclusion relies on the intuition that solving the Exact Candidate Query is usually more costly (i.e., requires more tests) then solving the Single Candidate Query. Below, we back that intuition.

6.2.1.3 Usefulness of Diagnosis Queries

Even if the queries have a significant meaning in several diagnosis domains, they might still be of little use. Indeed, answering the Exact Candidate Query allows us to answer the other two queries. For instance, if we know the actual diagnosis, we

definitely know whether a given hypothesis $h \in \mathcal{H}$ is the actual diagnosis or not.

What really makes the queries useful is our intuition that solving the Exact Candidate Query is generally harder (i.e., it requires more tests) than solving the Single Candidate Query, for instance.

To prove this assumption, we need to manipulate tests. As discussed in Section 4.3.1.3, we simply consider that a test consists of a set of outcomes. Each outcome refutes (resp. confirms) some hypotheses.

A solution to a query is a sequence of tests such that after the execution of those tests, the query can be answered based on the set of remaining candidates.

It should not be surprising that a solution to the Exact Candidate Query is always a solution to any Single Candidate Query, see Proposition 6.2.

Proposition 6.2

Solving the Single Candidate Query for $h \in \mathcal{H}$ requires at most as many tests as solving the Exact Candidate Query.

Proof:

Assume the Exact Candidate Query can be solved with the sequence of tests T. It means that after the execution of the tests in T, we have a unique diagnosis candidate. Thus, after the execution of the tests in T, we can tell whether h is the actual diagnosis or not, simply by comparing it with the single candidate left. □

However, it is sometimes possible to solve the Single Candidate Query with much fewer tests than what is required to solve the Exact Candidate Query, see Proposition 6.3.

Proposition 6.3

Solving the Exact Candidate Query could require $n - 2$ more tests than solving the Single Candidate Query for $h \in \mathcal{H}$ (n being the number of tests available).

Proof:

Consider the n tests $t_1, t_2, \ldots, t_n$ and the n hypotheses $h_1, h_2, \ldots, h_n$. Assume each test t_i has two possible outcomes:

- *refutes only h_i or*
- *confirms only h_i, i.e., refutes every hypothesis except h_i*

Starting from $\Gamma = \{h_1, h_2, \ldots, h_n\}$*, we can solve the Single Candidate Query for any* h_i *with a single test, namely* t_i*. If* t_i *confirms* h_i*, by the definition of* t_i *it also refutes every other test.* h_i *is then the only candidate left, thus it must be the actual diagnosis. But, if* t_i *refutes* h_i*, then* h_i *cannot be the actual diagnosis.*

In comparison, we need $n-1$ *tests to be guaranteed to solve the Exact Candidate Query. In the worst case, the first* $n-2$ *tests would refute only one hypothesis each, leaving still two candidates. Thus, solving the Exact Candidate Query can require* $n-2$ *more tests than solving the Single Candidate Query.* □

6.2.2 Test Selection Strategies

Historically [30], test selection in diagnosis is simply performed using a greedy approach, as discussed in Section 4.3.2. Here, we set the foundations for the elaboration of a test selection theory within diagnosis.

First, we define some interesting properties for studying each test selection problem. Then, we study in detail the properties for the test selection problem related to the Single Candidate Query. We also briefly consider the Exact Candidate Query, in order to illustrate that the test selection problems underlying two different queries are indeed quite different (another strong argument supporting our query-based extension).

6.2.2.1 Properties

We consider six properties when studying the test selection problem of each query:

Solution Structure: What structure do we need in order to represent a solution? A set, an ordered sequence or a tree of tests?

Optimal Characterizability: Is it possible to define the optimal solution of every instance of a query?

Solvability: Is there an algorithm that will always return the optimal solution?

Solution Verifiability: Given a potential solution, what is the complexity of verifying whether it is really a solution?

Complexity: What is the complexity of obtaining the optimal solution?

Approximability: Can we approximate the optimal solution with some guarantees?

6.2.2.2 Single Candidate Query

Here we study the Single Candidate Query with respect to the six properties mentioned above. But before getting there, let us consider our assumptions on the nature of the tests. These assumptions hold in OS discovery.

6.2.2.2.1 Assumptions

Assumption 6.2
We assume that every test is fully defined. That is, for every $h \in \mathcal{H}$, there is at least one possible outcome θ_i^t such that $h \in \Gamma(\theta_i^t)$.

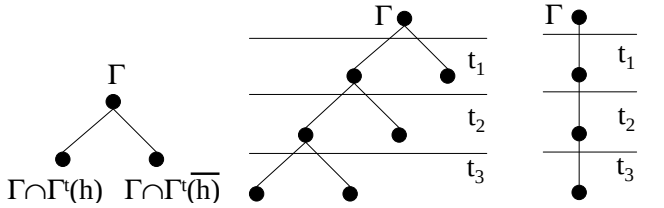
Definition 6.1 (Uniquely Supporting Test)
A test t is uniquely supporting if all its interpreted possible outcomes are pairwise disjoint. In other words, this means that for each hypothesis h, there is at most one possible outcome of t supporting h. ◯

Assumption 6.3
We assume all tests to be uniquely supporting, see Definition 6.1.

Note that from these two assumptions, it follows that each test leads to a partition of $\mathcal{H}$. Moreover, each test is deterministic in the sense that whenever it is executed in a situation where $\Delta \in \mathcal{H}$ is the actual diagnosis, the outcome will always be the same (the one supporting Δ).

Assumption 6.4
We assume all hypotheses to be pairwise distinct. That is, given any two hypotheses $h_1, h_2 \in \mathcal{H}$, there exists at least one test t such that the single outcome of t supporting h_1 does not support h_2.

Note that Assumption 6.4 can be enforced in the following way. If h_1 and h_2 are not distinct, then remove them from the system and add a new one h' representing them both simultaneously. Then it is possible to make this "wrapping" transparent to



(a) Single Test (b) Test Sequence (c) Simplified Test Sequence

Figure 6.5: Test Execution for Single Candidate Query

the user by "unwrapping" before answering queries. Such a method is important for OSD since, for instance, we are not aware of any test distinguishing between Windows 2000 sp2 and sp3.

Assumptions 6.2 and 6.3 lead to an interesting simplification for the Single Candidate Query. Assume we are interested in knowing whether h is the actual diagnosis in a situation where Γ is the current set of candidates. We can now see each test t as a bipartition $\langle S^t(h), S^t(\overline{h})\rangle$ of $\mathcal{H}$ where:

- $S^t(h) = \Gamma(\theta_i^t)$ such that $h \in \Gamma(\theta_i^t)$.
- $S^t(\overline{h}) = \cup\Gamma(\theta_i^t)$ such that $h \notin \Gamma(\theta_i^t)$.

The fact that each test is a bipartition of $\mathcal{H}$ for every $h \in \mathcal{H}$ makes reasoning about tests much easier for the Single Candidate Query. In fact, the execution of test t, in a situation where Γ is the current set of diagnosis, leads to a tree of height 1 rooted at Γ with two leaves: one $\Gamma \cap S^t(h)$ containing h and the other $\Gamma \cap S^t(\overline{h})$ not containing h (Figure 6.5(a)). Thus, with respect to the Single Candidate Query for h, we need only to consider what happens if we end up in the leaf supporting h, because if we end up in the other leaf, the answer to the query is *no*. When considering the execution of a sequence of tests $S = [t_1, t_2, t_3]$, we obtain a nearly linear tree of height 3 which is a solution only if one of its two deepest leaves is $\{h\}$ (Figure 6.5(b)). We will usually only consider the nodes containing h and omit the other ones, we thus obtain a fully linear tree (Figure 6.5(c)). Assumption 6.4 guarantees the existence of a solution for any instance of the Single Candidate Query.

From a reasoning point of view, the notion of discriminant power of a test will be crucial.

Definition 6.2 (Discriminant Power)
Given a test t, a set of diagnosis candidates Γ and a specific hypothesis h, the discriminant power of t for h with respect to Γ, denoted $discriminantPower(t, \Gamma, h)$, is the number of candidates from Γ that are eliminated by the outcome of t which confirms h. That is, $discriminantPower(t, \Gamma, h) = |\Gamma| - |\Gamma \cap S^t(h)|$. ◯

Proposition 6.4
Given an instance of the Single Candidate Query $\langle \Gamma, \mathsf{TEST}, h \rangle$, the discriminant power of any test $t \in \mathsf{TEST}$ can be computed in $O(|\mathsf{CONST}|^2)$.

Proof:
The time-consuming part of the algorithm is to compute the intersection $\Gamma \cap S^t(h)$. It is easy to see that the intersection of two sets A and B can be done in $O(|A| \times |B|)$. From there, it suffices to remark that both Γ and $S^t(h)$ are subsets of CONST (under the single fault model). □

6.2.2.2.2 Solution Structure We look at the solution structure from an off-line point of view, i.e., we first compute a solution and only then do we start executing tests. It is still possible to represent an on-line test selection process by making the choice of a test dependent on the outcome of the previous test. However, had we chose to look at test selection from an on-line point of view, the solution structure of any query would look the same and it would not have been possible to expose early differences in the queries (which we believe to be an indication of their different computational "difficulty").

A problem instance is defined by three components:

- A description of the initial situation. This corresponds to the current set of diagnosis candidates.
- A set of tests.
- A description of the goal. For the Single Candidate Query, the goal is to determine whether a single given hypothesis $h \in \mathcal{H}$ is the actual diagnosis.

The solution structure of the Single Candidate Query is simply a set of tests. Initially, we have a set of diagnosis candidates Γ and a hypothesis h to test. The result of each test will either refute h or confirm it. According to Figure 6.5, executing t from Γ will either lead to the solution (because the outcome of t refutes h) or to $\Gamma \cap S^t(h)$. So we need only to consider what happens in the latter case and which test we should then execute. It is straightforward to see that every ordering of the tests in a solution is also a solution.

6.2.2.2.3 Optimal Characterizability Given the solution structure of the Single Candidate Query discussed above, the optimal solution is simply the solution whose set is of smaller cardinality. Or the solution for which the sum of the cost of its tests is minimal.

6.2.2.2.4 Solvability There is a naive algorithm that will always find the optimal solution to the Single Candidate Query. Simply note that given a set of tests TEST, there are at most $2^{|\mathsf{TEST}|}$ possible solutions. Thus, we can: build them all, keep only the actual solutions, and finally, find the minimal one.

6.2.2.2.5 Solution Verifiability Given any instance of the Single Candidate Query $\langle \Gamma, \mathsf{TEST}, h \rangle$, we can verify a possible solution in polynomial time. Given a set of tests $T \subseteq \mathsf{TEST}$, compute $R = \Gamma \cap_{t \in T} S^t(h)$. T is a solution iff $R = \{h\}$.

6.2.2.2.6 Complexity We've shown that the problem is solvable and we've provided a simple, but exponential, algorithm. We will show below, in Theorem 6.1, that the problem is NP-Hard and thus intractable.

The optimization problem of the Single Candidate Query is: given a universe $\mathcal{H}$ (the hypothesis space), a set of tests TEST, and a specific hypothesis h, find the smallest solution to the Single Candidate Query for h, i.e., the solution containing a minimum number of tests.

The decision problem of the Single Candidate Query is: given $\langle \mathcal{H}, \mathsf{TEST}, h \rangle$ and an integer j, find whether or not there exists a solution to the Single Candidate Query of size j or less. We call this problem SingleCandidateQueryD.

By definition, SingleCandidateQueryD is a decision problem and to prove it belongs to NP we simply have to show that there is a solution certificate of polynomial size that can be verified in polynomial time. The solution certificate is a set of tests $T \subseteq \mathsf{TEST}$, clearly of polynomial size with respect to the problem size. We need to verify that there are at most j tests in T (easily done in polynomial time) and that $\cap_{t \in T} S^t(h) = \{h\}$ (again feasible in polynomial time, as each $S^t(h)$ has size of at most $|\mathsf{CONST}|$).

Below, we show that SingleCandidateQueryD is NP-Complete, making the optimization version NP-Hard. We do so by a reduction to the set cover problem. The set cover problem is known to be NP-Complete, see Section A.3 of [38].

Definition 6.3 (Set Cover Problem)
Given a universe $\mathcal{U}$ and a family $\mathcal{S}$ of subsets of $\mathcal{U}$, a set cover is a subfamily $C \subseteq \mathcal{S}$ of sets whose union is $\mathcal{U}$. Decision problem (SetCoverD): given a universe $\mathcal{U}$, a family $\mathcal{S}$ of subsets of $\mathcal{U}$, and an integer k, the question is whether there is a set cover of size k of less. ◯

Theorem 6.1 (NP-Completeness of SingleCandidateQueryD)
SingleCandidateQueryD is NP-Complete.

Proof:
We will show that SetCoverD $\leq_m^p$ SingleCandidateQueryD. This will be sufficient to show that SingleCandidateQueryD is NP-Complete as we already know that SetCoverD is NP-Complete. Given a universe $\mathcal{U}$, a family $\mathcal{S}$ of subsets of $\mathcal{U}$, and an integer k, we want to know if $\langle \mathcal{U}, \mathcal{S} \rangle$ has a set cover of size k or less. We have to construct a universe $\mathcal{H}$, a set of tests $\mathcal{T}$, an element h and an integer j such that:

- *the construction takes a polynomial time (with respect to the size of $\langle \mathcal{U}, \mathcal{S}, k \rangle$).*
- *$\langle \mathcal{U}, \mathcal{S} \rangle$ has a set cover of size k or less iff $\langle \mathcal{H}, \mathsf{TEST}, h \rangle$ has a solution to the Single Candidate Query of size j or less.*

Reduction:

- $j = k$
- $h =$ *"h", an element not in* $\mathcal{U}$

- $\mathcal{H} = \mathcal{U} \cup \{h\}$
- *TEST: For each set $X \in \mathcal{S}$, there is a test t_X. t_X is such that $S^{t_X}(h) = \mathcal{H} \setminus X$, while $S^{t_X}(\overline{h}) = X$. All tests are then uniquely supporting.*

Clearly, the construction takes polynomial time.
$\Rightarrow$[4]*:*
Assume $\langle \mathcal{U}, \mathcal{S} \rangle$ has a set cover $C \subseteq \mathcal{S}$ of size $k' \leq k$. Then, we claim that $T = \{t_X | X \in C\}$ is a solution to the Single Candidate Query for $\langle \mathcal{H}, \mathsf{TEST}, h \rangle$. It has size no greater than j as $|T| = |C| = k' \leq k = j$. So let us explain why T is a solution to the Single Candidate Query. Since C is a set cover for $\mathcal{U}$, we know that $\cup_{X \in C} X = \mathcal{U}$. This means that for every $e \in \mathcal{U}$, $e \in X$ for some $X \in C$, let us denote this set by $X(e)$. But, by the definition of TEST, we know that $e \in S^{t_{X(e)}}(\overline{h})$. Since $S^{t_{X(e)}}(\overline{h})$ and $S^{t_{X(e)}}(h)$ form a partition of $\mathcal{H}$, we know that $e \notin S^{t_{X(e)}}(h)$. Thus, for each $e \in \mathcal{U}$, there is a test $t \in T$ such that $e \notin S^t(h)$. From there, we conclude that $\cap_{t \in T} S^t(h) = \{h\}$.
$\Leftarrow$[5]*:*
Assume $\langle \mathcal{H}, \mathsf{TEST}, h \rangle$ has a solution T to the Single Candidate Query of size $j' \leq j$. Then, we claim that $C = \{X | t_X \in T\}$ is a set cover for $\mathcal{U}$. Clearly, it has size no greater than k as $|C| = |T| = j' \leq j = k$. So we have to explain why C is a set cover for $\mathcal{U}$. Since T is a solution to the Single Candidate Query, it means that $\cap_{t \in T} S^t = \{h\}$. In other words, for every element e in $\mathcal{H} \setminus \{h\} = \mathcal{U}$, there is a test t such that $e \notin S^t(h)$. Since $S^t(h)$ and $S^t(\overline{h})$ form a partition of $\mathcal{H}$, we know that for every $e \in \mathcal{U}$ there is one test $t \in T$ such that $e \in S^t(\overline{h})$. By the definition of our reduction, this means that for every $e \in U$, there is one set $X \in C$ such that $e \in X$ (if $e \in S^{t_X}(\overline{h})$, then $e \in X$). Thus we conclude that $\cup_{X \in C} X = \mathcal{U}$, which means that C is a set cover of $\mathcal{U}$. □

6.2.2.2.7 Approximability We start by reviewing the basic notions of approximation algorithms.

[4]To prove: If $\langle \mathcal{U}, \mathcal{S} \rangle$ has a set cover of size k or less, then $\langle \mathcal{H}, \mathsf{TEST}, h \rangle$ has a solution to the Single Candidate Query of size j or less.

[5]To prove: If $\langle \mathcal{H}, \mathsf{TEST}, h \rangle$ has a solution to the Single Candidate Query of size j or less, then $\langle \mathcal{U}, \mathcal{S} \rangle$ has a set cover of size k or less.

6.2.2.2.7.1 Approximation Algorithms Let P be an optimization problem and $opt(p)$ be the cost of the optimal solution for an instance p of P. When P is intractable, we might rely on an approximation algorithm. An approximation algorithm A for P is a polynomial time algorithm which might fail to return the optimal solution but does return a solution *close* to the optimal one. We use $A(p)$ to denote the solution returned by A on p.

In this thesis, we consider three classes of approximation algorithms, each providing its own definition of *close*, see [42].

- A is a c+approximation if $\forall p \in P$, $A(p) \leq opt(p) + c$, for a constant c. For instance, edge coloring in a graph is 1+approximable, see Chapter 9 or [42].
- A is a $c*$approximation if $\forall p \in P$, $A(p) \leq c \times opt(p)$, for some constant c. For instance, the knapsack problem is 2$*$approximable, see Chapter 13 of [11], and bin packing is 11/9$*$approximable, see [44].
- A is a $f(n)*$approximation if $\forall p \in P$, $A(p) \leq f(n) \times opt(p)$, for some function $f(n)$ where n is some parameter of the instance. For instance, vertex coloring in a graph is $n/\log n*$approximable (where n is the number of vertices), see [48].

Ideally, we want a c+approximation. Otherwise, a $c*$approximation is preferable to a $f(n)*$approximation.

6.2.2.2.7.2 Equivalence to Set Cover Before studying its approximability, we show that the Single Candidate Query is equivalent to the set cover problem. As a result, the approximability results known for the set cover will directly apply to the Single Candidate Query as well.

One has to be careful when using equivalence to transpose approximation results. Indeed, it is not sufficient that two problems be polynomially Turing equivalent[6]. A stronger notion of equivalence, L-equivalence, is required here. For instance, although minimum-size vertex cover $\equiv_T^p$ maximum-size independent set (see [33]), vertex cover is 2$*$approximable (see [40]) while independent set is not $c*$approximable for any constant c unless P = NP (see [9]).

[6]Each is polynomially Turing reducible ($\leq_T^p$) to the other.

Definition 6.4 (L-Reduction [4])
Let P and Q be optimization problems and c_P and c_Q their respective cost function. An L-Reduction of P to Q is a pair of functions $\langle R, S\rangle$ such that:

- *if x is an instance of P, $R(x)$ is an instance of Q.*
- *if y is a solution to $R(x)$, then $S(y)$ is a solution to x.*
- *there exists a positive constant α such that*

$$opt(R(x)) \leq \alpha opt(x)$$

- *there exists a positive constant β such that*

$$|opt(x) - c_P(S(y))| \leq \beta |opt(R(x)) - c_Q(y)|$$

We write $P \leq_L Q$ to denote the fact that P is L-reducible to Q. ◯

The impact of an L-Reduction between problems is stated in Lemma 6.1.

Lemma 6.1
If $P \leq_L Q$ and there exists a ϵ-approximation[7] for Q then there also exists a δ-approximation for P where

$$\delta = \epsilon\alpha\beta$$

with α and β being defined in the L-Reduction between P and Q.
Proof:
See proof of Proposition 7 in [27]. □

corollary 6.1
If $P \equiv_L Q$ and there exists a ϵ-approximation for P, then there also exists a ϵ-approximation for Q (i.e., they both have the same approximation ratio).

Here we are interested in the L-equivalence of set cover and the Single Candidate Query.

[7] An ϵ-approximation can be any of c+approximation, c∗approximation, or $f(n)$∗approximation

Theorem 6.2

SetCover $\equiv_L$ *SingleCandidateQuery.*

Proof:

The reduction proposed in the proof of Theorem 6.1 provides most of the L-Reduction. R *is the reduction itself,* S *is the mapping from a test to a set, as defined in the reduction. The constants* α *and* β *are both 1 because* $opt(x) = opt(R(x))$ *and* $c_P(S(y)) = c_Q(y)$. □

We can now transpose any approximability result for the set cover to the Single Candidate Query. In the rest of this section, we provide a few of those approximability results. We focus on the set cover problem, mainly because it is more intuitive and has been widely studied.

6.2.2.2.7.3 Basic Cases Given the Single Candidate Query for $h \in \mathcal{H}$, the greedy approach (Figure 6.6) selects the test t with the maximal discriminant power (see Definition 6.2). This algorithm has a worst case time complexity of $O(|\mathsf{TEST}|^2 \times |\mathsf{CONST}|^2)$. To see this, note that for every iteration of the while loop (2), we either stop (2.1.1 and 2.4.1) or we remove one test from TEST (2.6). Thus we loop through step 2 at most $|\mathsf{TEST}|$ times. Each time we loop through step 2, we go through all the remaining tests (2.3) and for each test we compute its discriminant power. Since we start with $|\mathsf{TEST}|$ tests and we remove one each time, we compute the discriminant power of a test

$$\sum_{i=1}^{|\mathsf{TEST}|} i = \frac{|\mathsf{TEST}| \times (|\mathsf{TEST}| + 1)}{2} \in O(|\mathsf{TEST}|^2)$$

times, in the worst case. Since computing the discriminant power of a test requires $O(|\mathsf{CONST}|^2)$ in the worst case, see Proposition 6.4, the algorithm of Figure 6.6 has a worst case complexity of $O(|\mathsf{TEST}|^2 \times |\mathsf{CONST}|^2)$.

Example 6.2 illustrates that the greedy algorithm is an unbounded approximation.

Example 6.2 (the greedy algorithm is an unbounded approximation)
Consider a situation where $\mathcal{H} = \{h_0, h_1, \ldots, h_{2^k}\}$ and the Single Candidate Query for h_0 (we start with $\Gamma_0 = \mathcal{H}$). Consider also the following $k + 1$ uniquely supporting tests:

GreedyTestSelection(TEST,Γ,h)

Provides the set of tests to execute to answer the Single Candidate Query

Input: TEST: the set of tests available
Γ the current set of diagnosis candidates
h: the hypothesis to isolate

Output: The set of tests to execute

---

1 $T \leftarrow \emptyset$
2 WHILE $\Gamma \neq \{h\}$
2.1 IF $|\mathsf{TEST}| = 0$
2.1.1 RETURN "no solution"
2.2 $t \leftarrow \text{argmin}_{t' \in \mathsf{TEST}}$ discriminantPower(t',Γ,h)
2.4 IF discriminantPower(t,Γ,h) = 0
2.4.1 RETURN "no solution"
2.5 $T = T \cup t$
2.6 $\mathsf{TEST} = \mathsf{TEST} \setminus \{t\}$
2.7 $\Gamma = \Gamma \cap S^t(h)$
3 RETURN T

Figure 6.6: Greedy Algorithm for Test Selection

t_a: $S^{t_a}(h_0) = \{h_0\} \cup \{h_i | i \text{ is odd}\}$

t_b: $S^{t_b}(h_0) = \{h_0\} \cup \{h_i | i \text{ is even}\}$

t_1: $S^{t_1}(h_0) = \{h_0\} \cup \{h_1, h_2\} \cup \{h_5, \ldots, h_{2^k}\}$

t_2: $S^{t_2}(h_0) = \{h_0\} \cup \{h_1, \ldots, h_4\} \cup \{h_9, \ldots, h_{2^k}\}$

t_i: $S^{t_i}(h_0) = \{h_0\} \cup \{h_1, \ldots, h_{2^i}\} \cup \{h_{2^{i+1}+1}, \ldots, h_{2^k}\}$

t_{k-1}: $S^{t_{k-1}}(h_0) = \{h_0\} \cup \{h_1, \ldots, h_{2^{k-1}}\}$

The optimal solution is $\{t_a, t_b\}$ and has size 2. The solution provided by the greedy algorithm could be of size $k+1$ (it includes all tests). In state Γ_0, three tests (t_a, t_b, t_{k-1}) have the highest discriminant power (see Table 6.3); assume[8] the greedy algorithm selects t_{k-1}. In the resulting state, $\Gamma_1 = \Gamma_0 \cap S^{t_{k-1}}(h_0)$, again three tests have the

[8]We could build an example where t_a and t_b never have the highest discriminant power until they are the only tests left. However, such an example would be tedious and harder to understand.

Table 6.3: Discriminant Power

Candidates / **Tests**	Γ_0	Γ_1	Γ_{i-1}	Γ_{k-1}	Γ_k	Γ_{k+1}
t_a	2^{k-1}	2^{k-2}	2^{k-i}	2	1	1
t_b	2^{k-1}	2^{k-2}	2^{k-i}	2	1	0
t_1	2	2	2	2	0	0
t_2	4	4	4	0	0	0
t_{k-i}	2^{k-i}	2^{k-i}	2^{k-i}	0	0	0
t_{k-2}	2^{k-2}	2^{k-2}	0	0	0	0
t_{k-1}	2^{k-1}	0	0	0	0	0

highest discriminant power; assume the greedy algorithm selects t_{k-2}. This process, i.e., $\Gamma_i = \Gamma_{i-1} \cap S^{t_{k-i}}(h_0)$, will continue until $\Gamma_k = \{h_0, h_1, h_2\}$ where t_a and t_b are the only tests that have not been executed. The greedy approach will then select successively t_a and t_b to finally provide a solution containing all k tests. $\diamondsuit$

The key idea of the above example is that increasing the value of k by one (adding one test and $2^{k+1} - 2^k$ hypotheses) increases the size of the greedy solution by 1, while the optimal solution remains the same. Thus, we can build an instance where the greedy solution is arbitrarily bad. As a result, we conclude that there is no constant c such that the greedy algorithm provide a c+approximaton or even a $c*$approximation. This is a strong argument against the claim [30] that one step lookahead is good enough for test selection in diagnosis, see the discussion in Section 4.3.2.

Assume an instance of the set cover where each element belongs to exactly one set. Then the optimal solution has to include every set. Obviously, the greedy approach will provide the same solution. Thus, in that case, the greedy approach returns the optimal solution.

The dual case yields the same result. Assume that each element now belongs to every set except one. Then the optimal solution needs exactly two sets and so does the greedy approach.

6.2.2.2.7.4 Literature Results The general set cover problem is known to be $\log_2 n*$approximable where n is the number of elements to cover [44]. We know [8] that it is not $c*$approximable for any constant c unless P = NP. We also know

that it is unlikely to be approximated to anything better than $\log_2 n$ [35]. This approximation ratio corresponds to the worst case scenario of the greedy approach presented in Figure 6.6.

A k-exact instance of the set cover problem is an instance where every element belongs to exactly k sets. In Section 6.2.2.2.7.3, we determined that any 1-exact instance is solvable optimally in polynomial time. A 2-exact instance is 2∗approximable. To see this, notice that any 2-exact instance is L-equivalent to an instance of the vertex cover: each set represents a vertex and there is an edge between two vertices if their respective sets share an element. Since the vertex cover is 2∗approximable, see Chapter 17 of [59], so is any 2-exact instance of the set cover.

A k-bounded instance of the set cover problem is an instance where every element belongs to at most k sets. k-bounded instance are k∗approximable [41]. Achieving this approximation ratio requires a strategy different from the greedy one [6].

6.2.2.2.7.5 Other Results

Theorem 6.3

Given an instance $\langle \mathcal{U}, \mathcal{S} \rangle$ of the set cover problem, let $n = |\mathcal{U}|$ be the number of elements to cover and m be the size of the largest set in $\mathcal{S}$. Then, the problem is $(n - m - 1)$+approximable.

Proof:

Start by selecting the largest set. Then only $n - m$ elements remain to be covered. This will require no more than $n - m$ sets (we use 1 set to cover each remaining element), giving a total of $n - m + 1$ sets. The optimal solution needs at least 2 tests. So the difference between the optimal solution and our approximation is no more than $n - m - 1$. □

6.2.2.3 Exact Candidate Query

Here we take a brief look at the properties of the test selection problem for the Exact Candidate Query.

6.2.2.3.1 Solution Structure A simple set structure is insufficient for the Exact Candidate Query. At any time, the execution of a test t can leave us in several

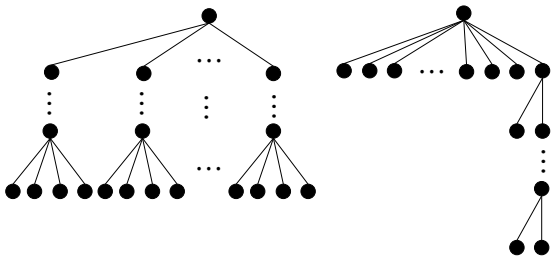


Figure 6.7: Tree Comparison Metrics

different situations. For instance, we could end up with $\Gamma \cap \Gamma(\theta_1^t)$ or with $\Gamma \cap \Gamma(\theta_2^t)$. By themselves, these are two different problem instances and each can have its own solution. For instance, in the former case it might be better to execute t_1 while t_2 is preferable in the latter. As a result, the solution must be structured according to a tree. That is, we start by executing the test at the root; then, given the outcomes to the tests executed so far, we walk down the corresponding path in the tree to obtain the next test to execute.

6.2.2.3.2 Optimal Characterizability We need to be able to compare two trees and decide which one is better. There are several possible metrics for such a comparison, but unfortunately none of them jumps out as being a natural choice. We could choose the tree of minimal height (worst case), or the tree having a leaf of minimal depth (best case), or the tree with the best leaf-depth average. However, none of these metrics provide an adequate characterization of the optimal solution. For instance, when choosing the tree of minimal height, we would choose the tree on the left in Figure 6.7. However, this is arguably not the best choice.

6.2.2.3.3 Solvability Unless we can formally define the concept of optimal solution for the Exact Candidate Query, it is impossible to consider whether there is an algorithm that will always provide us the optimal solution.

6.2.2.3.4 Solution Verifiability Given any instance of the Exact Candidate Query, that is $\langle \Gamma, \mathsf{TEST} \rangle$, we can also verify a possible solution in polynomial time. Given a tree of tests T, there are at most $|\Gamma|$ leaves. Thus verifying the solution simply requires computing the set corresponding to each leaf (each leaf requires computing

the intersection of at most height(T) sets) and verifying that each one is a singleton. This can be done in polynomial time.

6.2.2.3.5 Complexity It is not possible to establish the complexity of finding the optimal solution for the Exact Candidate Query until we have a formal characterization of the optimal solution.

6.2.2.3.6 Approximability Again, it is not possible to discuss approximation since we do not have an optimal solution to compare with.

6.2.2.4 Summary

Our two key contributions to diagnosis are: a query-based extension to provide more flexibility and the foundations of a theory of test selection.

The query-based approach has been shown to be both meaningful and useful in different diagnosis domains.

The results we have provided regarding test selection are important in the following ways:

- The queries are now known to lead to different computational problems, another argument in favor of a query-based approach to diagnosis.

- The classical one-step lookahead strategy for test selection is now known to not be good enough in the general case.

So far, we have addressed two of the three main objectives of this thesis; that is, design a better tool for OSD (that would be our hybrid approach, it will be shown to provide better experimental results in the next chapter), and give our tool a strong theoretical background (that would be the theory of diagnosis). One objective still remains to be addressed: a systematic way of gathering OS fingerprints. This final objective is discussed in the following section.

6.3 Virtual Network Experiment Controller

The advantage of having a systematic and automated way of gathering OS fingerprints is to avoid relying on users to submit their own ad hoc fingerprints. However, it would not be practical to dedicate one computer for every existing OS just to have them available for fingerprinting purposes. Instead, we rely on virtualization technologies (like VMWare [75, 76] and VirtualPC [49]) and we fingerprint virtual machines (VMs). This allows us to experiment with hundreds of different operating systems at a very low cost.

As mentioned in Section 2.2.1, OSD tests are either passive, active or both. Tests that are entirely passive are based on spontaneous events, i.e., events that must be initiated by the computer itself or its user and cannot be triggered remotely on-demand (i.e., by sending a specific stimulus packet). Active test, on the other hand, are based on reactive events, i.e., events occurring in response to a stimulus. Since reactive events do not require any action from the fingerprinted VM, we simply need to stimulate it remotely. Spontaneous events, however, must be initiated by the fingerprinted VM itself. Thus we have to be able to control[9] every VM in order to have them generate those spontaneous events.

In this Section, we present VNEC [14] (virtual network experiment controller), a tool to automatically execute experiments, such as fingerprint gathering, in a virtual environments. A prototype of VNEC is available from `vnec.sourceforge.net`. First, Section 6.3.1 describes a generalized version of the OS fingerprint gathering problem. Section 6.3.2 presents VNEC and Section 6.3.3 explains how it can be used for OS fingerprint gathering. More information about VNEC can be found in [16, 17].

6.3.1 The General Problem

OS fingerprinting is one type of experiment that can be executed in a virtual environment. The benefits are that we can study many different OSes at a low cost and the process can be fully automated. However, other experiments would benefit from a general tool automating their execution in a virtual environment. Examples are:

[9] We want to perform the fingerprint gathering automatically.

- Studying the spreading patterns of viruses.
- Analyzing the behavior of different targets with respect to some given attacks.

We want VNEC to be able to support these experiments as well.

Performing these experiments in a physical network would be time consuming, since the computers need to be cleaned after each virus attack, and expensive, since we need several physical machines to host a wide variety of OSes. If we are to perform these experiments in a virtual environment, the environment must support the following requirements:

- The environment must be confined, to make sure the effects of security sensitive experiments do not spread to the physical machine(s) hosting the experiment.
- It has to provide a wide variety of guest OSes.
- The environment must be able to control every VM to allow the generation of spontaneous events (for OS fingerprinting) and client-to-server attacks (for attack reaction study, see Example 6.3).

Example 6.3 (client-to-server attack scenario in a virtual environment) Figure 6.8 illustrates the scenario of a client-to-server attack (in a virtual environment). Initially, we want the client VM to initiate a connection with the malicious server and request content (step 1). Then, the server replies with malicious content possibly compromising the client (step 2). Finally, the sniffer records the traffic generated by the client as a reaction to the malicious content (step 3). We want to replay this experiment many times using different client VMs to see how different OSes react to the same malicious content. Thus, the VMs that need to be controlled are: the sniffer and all the clients. Controlling a VM means forcing it to execute a specific task, e.g., request content from the malicious server. ♢

The example above can also be seen from our OS fingerprinting point of view. Instead of a client, we talk about the fingerprinted host, and instead of requesting content, it generates a spontaneous event (e.g., echo request or TCP SYN packet). Thus, our tool must be able to force any VM to execute a specific task.

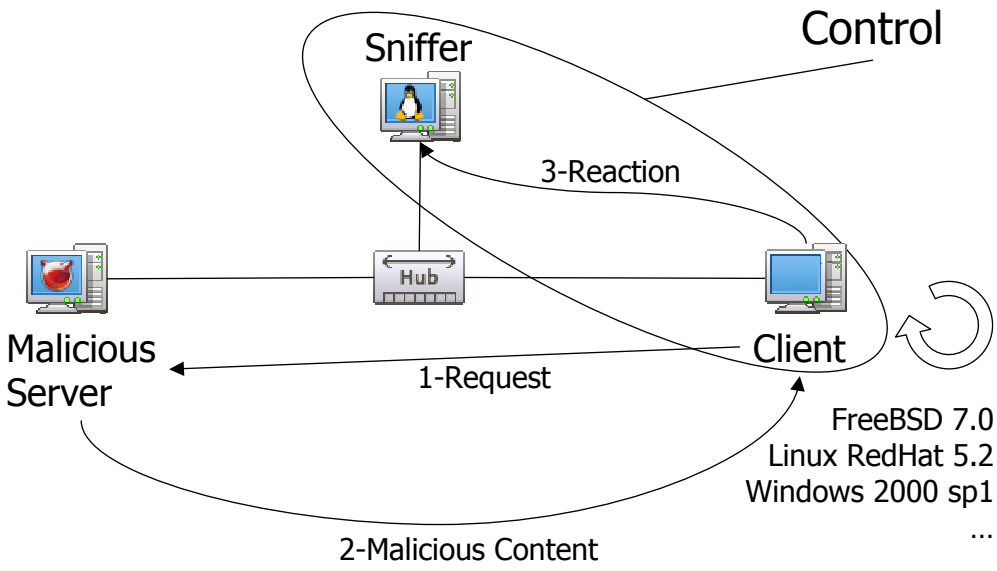


Figure 6.8: Client-to-Server Attack Experiment

6.3.2 VNEC Architecture

VNEC has three modules, they are detailed individually below:

- The network specification module, which defines the network topology and the set of VMs to use.
- The task workflow specification module, which specifies tasks to be executed by the VMs and their order.
- The experiment execution module, which is used to configure the VMs and dispatch the tasks in the desired order.

6.3.2.1 Network Specification

Figure 6.9 depicts the network specification graphical interface of VNEC. The network specification phase consists of:

- Creating the set of components (computers, hubs, and routers) using drag-and-drop.

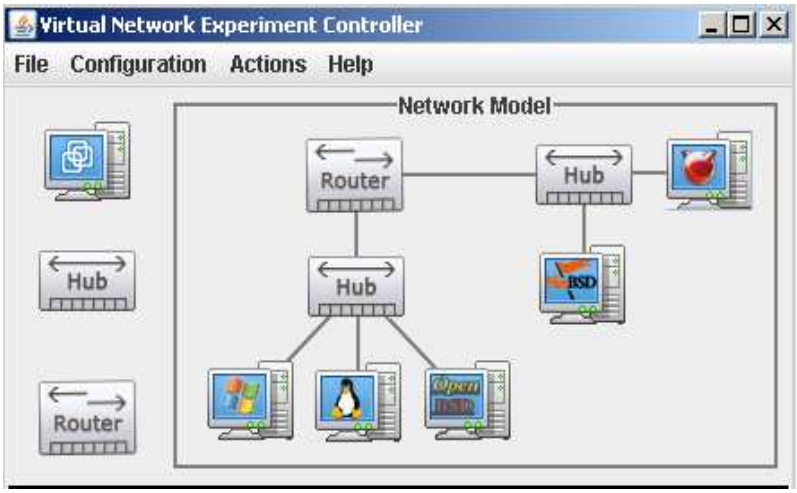


Figure 6.9: Snapshot of VNEC - Network Specification

- Specifying the network topology by connecting the components according to some rules (e.g., computers cannot directly connect to other computers, each computer has at most one connection).

- Associating each computer to a virtual machine (from a set of pre-existing VMs).

In Figure 6.9, the user connected five machines (in clockwise order from top-right: FreeBSD NetBSD, OpenBSD, Linux, and Windows) using two hubs and a router. The network consists of two segments: The FreeBSD and NetBSD hosts form one segment and the other three hosts form the other. Routers are implemented using dedicated VMs, while hubs are the default behavior on a virtual segment (VMWare VMNet and VirtualPC loopback adapter). By clicking on a computer icon, it is possible to select the snapshot to use for the specific VM (for VMWare only). By default, the current snapshot is used.

6.3.2.2 Task Workflow Specification

The task workflow fulfills two roles: it allows the user to indicate which tasks should be executed by the virtual machines and to specify the order of execution. The task

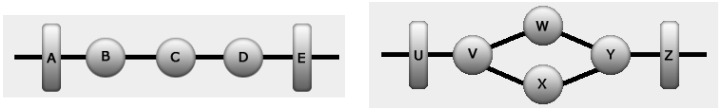
workflow is a directed acyclic graph [13] with a single source and a single sink[10] where each node corresponds to a task (Figure 6.10). The semantics of such a workflow is that a task is to be executed when all its parents are completed.

A task is either a *command task* or a *control task*. Command tasks are executed by a virtual machine (e.g., *create file*, *delete file*, *kill process*, *open telnet connection*, *open web browser*, etc.), while control tasks are performed by the controller to modify the state of a virtual machine (e.g., *power on*, *shut down*, *take snapshot*, *revert to snapshot*, *clone*, etc.). One must assign a task to each node; this can be done in a custom way by providing the set of command strings that must be executed or by selecting and configuring a predefined command template. A command template usually requires some parameters; for instance, the delete file command template requires a file name. Each command template will be automatically converted into command strings at run time by the VM. For instance, the command structure to delete the file "`name.txt`" would translate to "`rm -f name.txt`" on a Linux VM and to "`del name.txt`" on a Windows VM.

A task workflow reads from left to right; a circle represents the execution of a task by a single given VM, while a rectangle stands for the execution of a task by a group of VMs. For instance, the task workflow shown in Figure 6.10(a) starts with task A which is executed by all VMs (e.g., power on). Once task A is completed, task B, is performed by a single VM (e.g., Linux starts recording traffic). Afterwards, task C (e.g., OpenBSD launches a specific attack against Windows) and task D (e.g., Linux stops recording the traffic) are executed in sequence. Finally, E (e.g., power off) is applied on all virtual machines.

A task workflow does not have to be linear as displayed in Figure 6.10(b). In this case, once task V is completed, both tasks W and X begin concurrently. Task Y will start only after both W and X are completed.

[10] A source (resp. sink) is a node with no incoming (resp. outgoing) edges, i.e., a root (resp. leaf).



(a) Linear Task Workflow (b) Non-Linear Task Workflow

Figure 6.10: Examples of Task Workflows

6.3.2.3 Experiment Execution

Once both the network and the task workflow are specified, the experiment is ready to be launched. To be able to dispatch commands to any virtual machine, we implemented two mechanisms to communicate with them: through shared folders (using the VMWare shared folder feature) and through remote method invocation (using Java RMI). Moreover, we rely on a special Linux VM dedicated to dispatching commands from the controller (i.e., the physical host) to any VM, we call it the *dispatcher*.

The VMWare shared folder feature allows the physical host and the virtual machines to access a common folder. A VM can simply look for a specific file in the shared folder, parse it and interpret its content. To dispatch a command to a specific VM, the controller creates a file representing the command and places it in the shared folder to be processed by the corresponding VM. This process is both simple and safe[11]. However, the shared folder feature is available only for some virtual machines[12]. To circumvent this limitation, we developed a second mechanism.

In the second mechanism, each virtual machine is running the server side of a Java RMI application; another VM can call the function `execute(Command c)` on the server. Unfortunately, the controller (the physical machine) cannot communicate directly with the VM through the network (for safety reasons). To address this problem, we include a Linux VM dedicated to dispatching commands, *dispatcher* in Figure 6.11. The controller tells the dispatcher, through the shared folder control link, which task should be executed by which virtual machine. Then, the dispatcher forwards the task to the corresponding VM through the Java RMI control link.

[11]It is safe because it allows to isolate the physical host from the virtual network and thus prevents a virus to spread outside the virtual environment.

[12]It requires the VMWare tools to be installed on the VM, and this can only be done with recent versions of Windows and Linux.

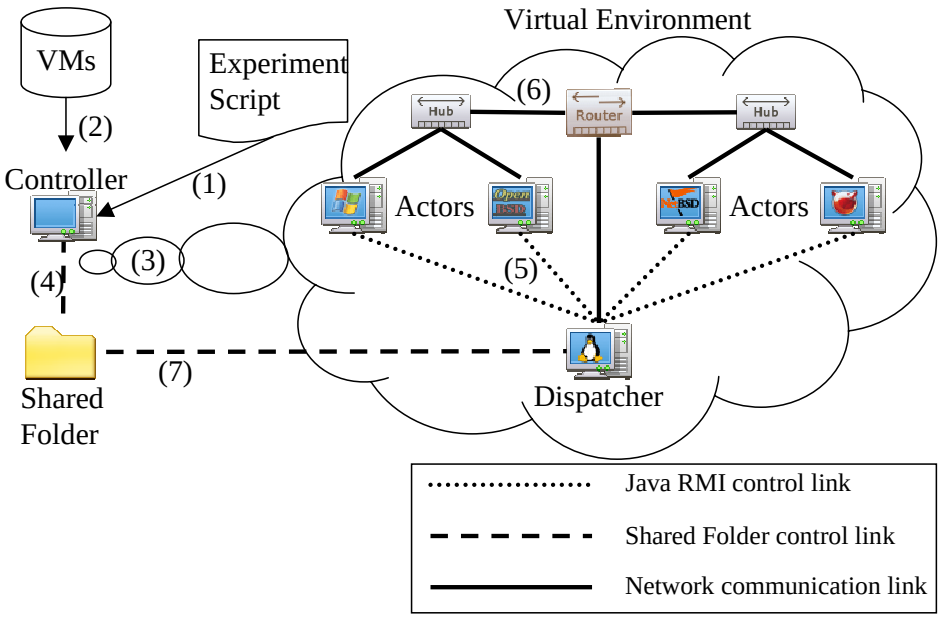


Figure 6.11: VNEC Communication Architecture

The two mechanisms are used together to dispatch commands to any VM while still providing a strong containment of the virtual environment.

As depicted in Figure 6.11, VNEC works as follows:

- The user provides an experiment which consists of a network and task workflow specification (step 1 of Figure 6.11).

- The controller fetches the specified VMs from a repository (step 2) and then creates the virtual network (step 3).

- Once the network is ready, the controller takes the first task to be executed and asks the dispatcher to send that task to the corresponding VM (step 4). This is done through the shared folder control link.

- The dispatcher sends the task to the given VM (step 5). This is done through the Java RMI control link.

- The VM executes the task (step 6).

- The dispatcher retrieves the task result, if any, from the VM and transfers it to the controller via the shared folder (step 7).

This architecture allows us to dispatch tasks to any virtual machine supporting Java. It is also general enough to be used for other network experiments, not just for OS fingerprinting.

6.3.3 OS Fingerprinting with VNEC

Here we explain how VNEC can be used to gather OS fingerprints. We provide two examples: one for gathering fingerprints based on reactive events and another for fingerprints based on spontaneous events. Moreover, we discuss the main limitation of our approach for gathering fingerprints.

6.3.3.1 Reactive Events

We first consider how to use VNEC to gather fingerprints for the TCP RstAck test (see Definition A.9). We are interested in the reaction of an OS to a SYN packet sent to a closed port. More specifically, we are interested in the DF and TTL value of the response (we know the response will be a TCP RST/ACK packet).

For a given OS, the response DF value can either be: always set, never set, or echoed (the same value as the DF in the stimulus packet). Moreover, the TTL value can either be a specific value between 1 and 255 or echoed. To make sure we capture all the possible cases, we send two stimuli:

- DF is set, TTL = 64
- DF is not set, TTL = 128

To run this in VNEC we use three VMs: the target to fingerprint, a sniffer to record the traffic, and a stimulator to send the two stimuli. We use a simple network topology where all the VMs are connected through a hub. Then, we perform the following tasks in this specific order:

- Power on all VMs.

- The sniffer starts recording traffic.
- The stimulator sends the two stimuli (on a closed port).
- Wait 5 seconds.
- The sniffer stops recording traffic.
- Retrieve the traffic trace from the sniffer.
- Power off all VMs.
- Start over with a different target to fingerprint.

Based on the resulting traffic traces, we can extract the fingerprints for every OS used in the experiment.

6.3.3.2 Spontaneous Events

Now we consider how to use VNEC to gather fingerprints for the TCP Syn test (see Definition A.1). We are interested in the way each OS builds their SYN packets when initiating a TCP session. More specifically, we are interested in the DF and TTL values of the SYN packets sent by each OS.

There are several situations in which a machine will send a SYN packet: to open a FTP connection, to open a web page, to initiate a SSH connection, to open a telnet connection, etc. We use FTP, SSH and telnet for all VMs. This allows us to obtain a reasonable sampling of SYN packets.

To run this in VNEC, we use three VMs: the host to fingerprint, a sniffer to record traffic and a dummy target for the host to try connect to. We again use a simple topology with a single network segment. Then, we perform the following tasks in this specific order:

- Power on all VMs.
- The sniffer starts recording traffic.
- The fingerprinted host tries to open a FTP connection on the dummy target.

- The fingerprinted host tries to open a SSH connection on the dummy target.
- The fingerprinted host tries to open a telnet connection on the dummy target.
- Wait 3 seconds
- The sniffer stops recording traffic.
- Retrieve the traffic trace from the sniffer.
- Power off all VMs.
- Start over with a different fingerprinted host.

6.3.3.3 Limitations

Although VNEC is a significant improvement over the current manual process for OS fingerprints gathering (not to mention that it is very general and can be used for several other network experiments), it is still not perfect. The main limitation of our OS fingerprint collection approach is not directly related to VNEC but to the idea of using a virtual environment. Current virtualization technologies are OS oriented. That is, they allow to run virtual instances of different operating systems. However, many networking devices (e.g., switches, printers, handheld devices, game consoles) run firmware instead of an OS. Since it is not possible to run virtual instances of firmware programs, our approach to fingerprint collection cannot fingerprint firmware. Network devices are important from a security point of view because, like OSes, they suffer from vulnerability (e.g., SecurityFocus BID 31092 lists the Apple iPhone as being vulnerable and BID 16954 provides a vulnerability for some Linksys routers). Moreover, like OSes they often have their own TCP/IP stack implementation and thus they have their own behavior (i.e., fingerprint); in fact, most OSD tools include some firware products in their database.

Since our fingerprint gathering technique does not work for firmware, we would have to rely on the same ad hoc process of manually collecting fingerprints for firmware. Currently, firmware fingerprints are not included in our tool.

6.4 Discussion

This chapter presented a theoretical model for a new approach to OS discovery. In theory, our approach has several advantages over classical ones (e.g., memory, multi-packets signature, a combination of active and passive tests, reasoning for test selection). It remains to be seen if these theoretical enhancements actually lead to practical improvements. The next chapter will verify this through an evaluation of our tool and a comparison to other OSD tools.

Chapter 7

Implementation & Evaluation

In theory, there is no difference between theory and practice. In practice, there is.
-Yogi Berra

The hybrid approach to OS discovery presented in the previous chapter promises significant improvement over current OSD tools. The objective of this chapter is to measure this improvement through an experimental evaluation.

The chapter first describes the implementation of our hybrid approach to OSD. Then, it presents experimental results comparing its accuracy with existing OSD tools.

7.1 Implementation

We discuss three topics related to the implementation of our tools: the OS fingerprints database, the passive module, and the active module. The tool is an open source project available from `http:\\hosd.sourceforge.net`.

7.1.1 OS Fingerprints Database

OS fingerprints can automatically be extracted from traffic traces (generated manually or with a tool such as VNEC). The fingerprints are stored in a database containing two tables for each OSD test. The first table contains all the fingerprints seen during extraction (e.g., Table A.1). The second table associates every OS to its corresponding fingerprint (e.g., Table A.2).

7.1.2 Passive Module

The passive module, i.e., candidate generation, is implemented using a combination of Prolog and Java algorithms. The Java module starts by gathering[1] packets using the jpcap library [12]. The relevant packets are transformed into predicates and written into a fact file. Once the number of packets written in a file reaches a threshold, or upon a user request, the Java module launches a Prolog process to compute the conflict sets, as per Definition 4.8. The threshold should be large enough to avoid breaking apart the packets forming an observation (e.g., a stimulus-response pair), but small enough so we don't clog the fact database to the point of slowing the conflict set computation. We currently use a threshold of 100.

Once the conflict sets are computed, the Java program computes the minimal hitting sets by intersecting the conflict sets, as discussed in Section 6.1.4.6. The result is the current set of possible OSes. At this point, the fact file is cleared and one entry is written to represent the current set of possible OSes. Below we provide more detail regarding the prolog part of the passive module.

7.1.2.1 Using Prolog

From the fingerprint database, three text files are generated for the passive module: possibleOSList.txt, passiveOSFingerprinting.pl, and allOSList.txt.

The file possibleOSList.txt lists the OSes handled by the tool. From this file we can generate the initial set of possible OSes.

The file passiveOSFingerprinting.pl contains the prolog rules used to compute the conflict sets based on given observations. The rules all have the form

$$\text{conflict\_set} \leftarrow \text{network\_event}$$

Network events are described as a conjunction of packet predicates, while the conflict sets are simply tags referring to a specific set of OSes (see discussion below regarding the allOSList.txt file). Figure 7.1 shows some entries of the passiveOSFingerprinting.pl file.

[1] Either from a pcap file or directly from the network.

```
set(X,16) :- tcp(X,Y,_,_,yes,"syn",64,16384,"M@1460",_,_).
set(X,2) :- arp(X,_,1,mac00_00_00_00_00_00).
set(X,5041) :- tcp(Y,X,Yp,Xp,DF,"syn",TTL,WIN,"M@265ST",SSNum,SAN),
        tcp(X,Y,Xp,Yp,no,"ack_syn",64,WINresp,"M@1460NNT",RSN,RAN),
        RAN is SSN + 1.
```

Figure 7.1: Content of passiveOSFingerprinting.pl

The file allOSList.txt contains the definition of all the sets of OSes. Each line contains the definition of one set, i.e., the OSes contained in the set. Line X contains the definition for set X and each line as the same length. Thus, the structure allows for random access to a specific set definition in constant time.

7.1.3 Active Module

The active module is entirely implemented in Java. The code is generic and the test definitions are loaded from a text file (testPossibleOutcomes.txt). This helps to prevent the modification of the Java code when updating existing tests with new OSes. However, when incorporating a new test, the Java code still needs to be modified: the tool has to know what stimuli to send for each test and this information is currently hardcoded. We are considering an alternative to circumvent this problem: a more elaborate test definition file dictating how to build the test stimuli.

The test definition file is automatically generated from the fingerprint database. Figure 7.2 shows some entries from testPossibleOutcomes.txt. Each line contains the definition of a test: its name and the set of OSes forming its possible outcomes. The name corresponds to one of the tests presented in Appendix A. Names containing a letter are subtests using different values in the stimuli packet (e.g., different TTL).

```
Test14: set(5057), set(5076), set(5102)
Test10a: set(5057), set(5075), set(5084), set(5085), set(5086),
        set(5087), set(5088), set(5090)
Test10b: set(5057), set(5075), set(5084), set(5085), set(5086),
        set(5087), set(5089)
```

Figure 7.2: Content of testPossibleOutcomes.txt

We currently consider three test selection strategies: the greedy one for the Exact Candidate Query, the greedy one for the Single Candidate Query, and the brute force one for the Single Candidate Query. They are described below.

7.1.4 Greedy - Exact Candidate Query

Given a set of currently possible OSes Γ and one possible outcome θ of a test, we compute the score of that outcome as shown in Equation 7.1.

$$score(\Gamma, \Gamma(\theta)) = |\Gamma \cap \Gamma(\theta)| - 1 \tag{7.1}$$

Intuitively, the score of an outcome is a measure of the distance between that outcome and a solution (0 meaning the outcome is a solution and -1 meaning the outcome is not possible). Based on this notion, we define the score of a test t with respect to a current set of possible OSes Γ as the average score of the possible[2] outcomes of t.

The greedy approach consists of selecting the test with the best (i.e., lowest) score.

7.1.5 Greedy - Single Candidate Query

As studied in Section 6.2.2.2, the Single Candidate Query has specific properties, especially when tests are uniquely supporting, as it is the case in OS discovery. Based on this, we use the greedy algorithm presented in Figure 6.6 with the notion of discriminant power provided in Definition 6.2.

Each test t has a single outcome supporting h, for any given h; we denote this outcome by θ_h^t. We then select the test t maximizing $|\Gamma| - |\Gamma \cap \Gamma(\theta_h^t)|$ (i.e., the discriminant power) as the next test to be performed.

7.1.6 Brute Force - Single Candidate Query

The brute force approach consists of selecting the smallest subset of tests which is a solution to the query. Given a subset of tests $T = \{t_1, t_2, \ldots, t_n\}$, T is a solution to the Single Candidate Query for h if the intersection of the interpreted outcome

[2]Not considering the outcomes with a negative score.

supporting h for every test is $\{h\}$, see Equation 7.2.

$$\Gamma(\theta_h^{t_1}) \cap \Gamma(\theta_h^{t_2}) \cap \ldots \cap \Gamma(\theta_h^{t_n}) = \{h\} \tag{7.2}$$

7.2 Experiment Results for Test Selection Strategies

Before we can compare our tool with existing OSD tools, we must decide which test selection strategy to use. As mentioned in Example 6.2 the one-step lookahead greedy strategy, selecting the test that is locally the most promising, is, in general, an unbounded approximation of the optimal solution. That is, it might require arbitrarily more tests than the optimal solution.

Before we decide which test selection strategy to use for our tool, let us see how good, or bad, is the greedy approach compared to a brute force approach leading to the optimal solution in our specific OS discovery context.

7.2.1 Experiment Setup

For this experiment, we consider the 95 targets available to us (see Appendix B) and we use the Single Candidate Query. We will consider 9025 cases: for each of the 95 targets and for each of the 95 different OSes included in the dataset, we will ask if the given target is running the given OS.

For each case, we use both the greedy and the brute force test selection strategies. The greedy strategy will select the test with the best score based on the definition provided in Section 7.1.5. The brute force strategy, on the other hand, will consider every possible subset of tests to find the solution with as few tests as possible, as described in Section 7.1.6. The idea is to see how far the greedy approximation is from the optional solution.

7.2.2 Results

Table 7.1 provides a classification of the 9025 cases into three categories:

- Cases for which the brute force approach is better (i.e., requires fewer tests) than the greedy one.

Table 7.1: Comparing Test Selection Strategies

brute force is better than greedy	45
brute force is equivalent to greedy	8958
brute force is worse than greedy	22

- Cases for which the brute force and the greedy approaches are equivalent (i.e., require the same number of tests).
- Cases for which the greedy approach is better than the brute force one.

For the vast majority of cases (8958 out of 9025), the two approaches are equivalent. Hence, it seems like the greedy approach is nearly optimal for our OSD problem.

There are only 45 cases for which the brute force approach requires fewer tests than the greedy one. While further inspecting those cases, we found out that the optimal solution was actually the same size as the one provided by the greedy strategy. What happened is that not all of the tests in the solution returned by the brute force strategy where executed. Our definition of a "solution", see Section 6.2.2.2.2, requires the underlying subset of tests to provide an answer to the query in any possible situation. However, given an optimal solution, not all of its tests need to be executed in every situation, see Example 7.1. Thus, simply due to a fortuitous ordering of the test by the brute force test selection, there are 45 cases where the greedy approach seems sub-optimal in practice, while in fact it is optimal for those cases as well (we simply didn't execute all tests in the optimal solution).

Example 7.1 (Test Ordering in a Solutoin)
Assume, for simplicity, that there are four possible OSes: A, B, C, D, and E and that we have two tests available: t_1 and t_2. Each test has two possible outcomes, as follows:

- t_1: $\{A, B\}$ and $\{C, D, E\}$
- t_2: $\{A, C\}$ and $\{B, D, E\}$

Now, assume we want to know if a computer is running the OS A. The optimal solution is $\{t_1, t_2\}$. The greedy approach, however, could decide to execute t_2 first

(after all, both tests have a score of 1 here). Now, further assume that the computer is actually running the OS C (note that we don't have this information during the test selection process). Then, after the execution of the first test in the optimal solution, namely t_1, we already know that the answer to the query is no and we don't need to execute t_2. The greedy approach, on the other hand, will end up executing both tests due to an unfortunate (non-deterministic) decision of choosing to execute t_2 first. Note that even though we ended-up executing only one test, the optimal solution truly has size 2 and thus the greedy approach is optimal. ♢

It might come as a surprise that for 22 cases, the greedy approach requires fewer tests than the optimal solution. But again here, the explanation is simply a fortuitous ordering of the tests. This time however, the greedy approach was luckier than the brute force.

7.2.3 Summary

Having considered all 9025 possible instances (for each of the 95 targets and for each of the 95 different OSes included in the dataset, we tested whether the greedy strategy is optimal), we can conclude that, for the current version of our OSD tool, the greedy test selection strategy is optimal up to test ordering (which we cannot control). As a consequence, in the rest of the experiments we consider only the greedy test selection strategy.

Note that this does not mean that the greedy approach is always optimal for every diagnosis problem. This specific result only applies to our particular OSD instance. This could change with the addition of a new OSD test or the update of those tests to include new OSes.

7.3 Evaluation Experiment Results

We reconsider the two experiments presented in Chapter 3: OS identification using the Exact Candidate Query and IDS alarm filtering using the Group Candidate Query. We now include the performance of the three modules of our own tool: posd, aosd, and hosd. posd is passive only, aosd is active only, and hosd combines both the passive

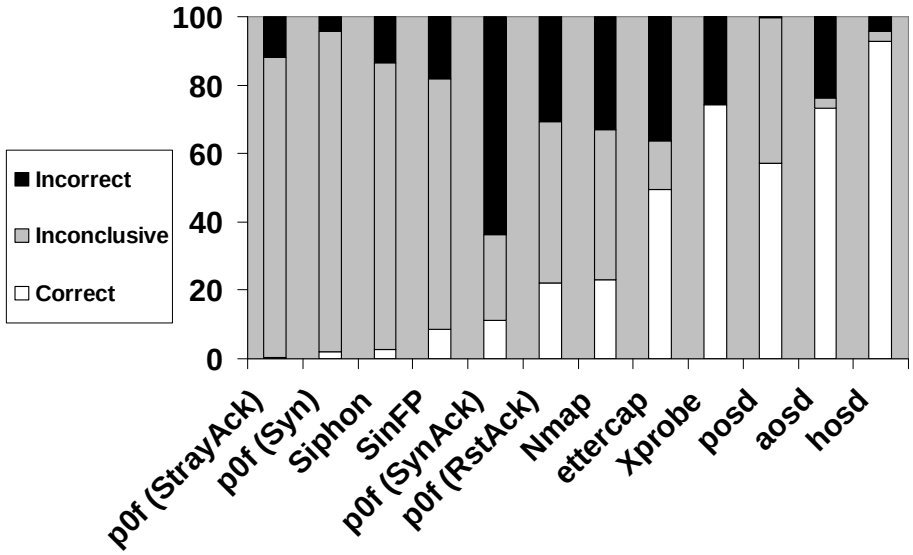


Figure 7.3: Correctness for the Exact Candidate Query

and active modules. This will allow us to evaluate the importance of each individual module in our hybrid approach and to compare the end result with other tools.

7.3.1 Exact Candidate Query

We start with the OS identification experiment initially presented in Section 3.2.4. We measure the correctness (how often does the tool provide a correct/ incorrect/ inconclusive answer) and imprecision (how large is the set of possible OSes for correct answers). We also measure the number of packets injected by active tools.

7.3.1.1 Correctness

Figure 7.3 presents the correctness for the three modules of our hybrid tool (also showing correctness of other OSD tools for comparison). Here is our interpretation of these results:

- posd is better than every other passive tool. This confirms our intuition that a knowledge-oriented design for OSD is useful. We believe the main reasons for the success of posd are its ability to model complex phenomena (stimulus-response) and the presence of a memory to keep previous information.

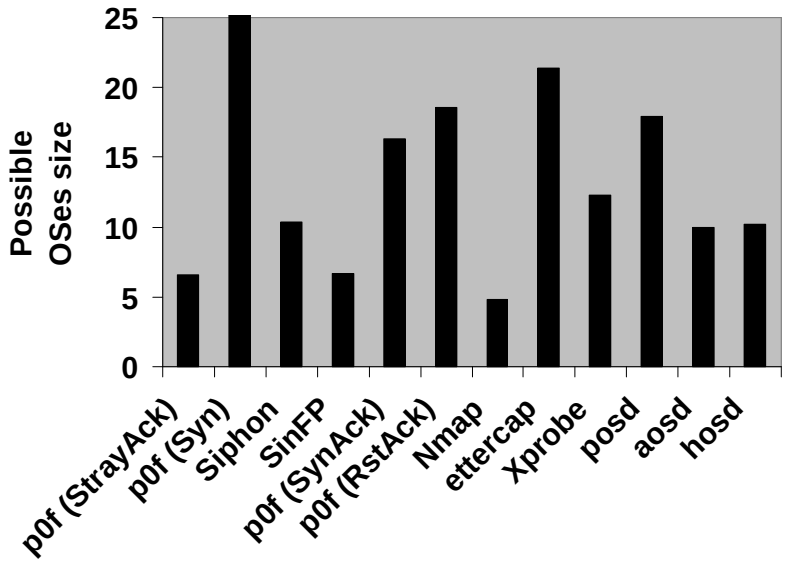


Figure 7.4: Imprecision for the Exact Candidate Query

- The combination of our passive and active modules provides excellent results. hosd is significantly better than any existing tool. It is also interesting to note that neither posd nor aosd are as good as hosd. This confirms our intuition that the hybrid approach combines the advantages of both the passive and active approaches. This improvement is due to the fact that active and passive tests are complementary; some phenomena cannot be triggered actively, while some others are rarely observed passively, see Section 2.2.1.

- posd and hosd rarely provide the wrong answer. On the other hand, aosd guesses wrong quite often (24%). This could indicate a problem in our active module, or more generally a limitation of the active approach.

7.3.1.2 Imprecision

Figure 7.4 provides the imprecision for the three modules of our tool (also showing imprecision for other OSD tools). Recall that the lower the imprecision, the better. Here are some interesting conclusions extracted from these results:

- hosd and aosd perform very well. The tools having the best precision all have a highly inadequate correctness. hosd is arguably the best tradeoff between precision and correctness.

Table 7.2: Packet Injection Summary for Single Candidate Query

	Nb packets sent		
Tool	***Min***	***Mean***	***Max***
Nmap	882	1686	2186
Xprobe	7	7	7
aosd	3	13.3	21
hosd	0	3.9	13

- posd does not have a very good precision by itself. Again, this confirms the importance of combining the passive and active approaches together.

7.3.1.3 Traffic Generated

The main concern with respect to active tools is the amount of traffic injected. Here, we take a look at the number of packets injected by active tools (including aosd and hosd) when trying to identify the actual OS, see Table 7.2. We observe the following:

- Nmap injects a high volume of packets. This is mainly due to the port scan performed before starting the OS discovery module. It is possible to disable this port scan, in which case Nmap still injects 30 packets on average, but the cost is an important decrease of accuracy. It is fair to mention that Nmap was not designed specifically to be an OS discovery tool; however, it is worth noting that **Nmap** is currently one of the most, if not the most, popular tool for OS discovery.

- The remaining three tools perform at a much better level. Xprobe is less intrusive than aosd, and hosd is less intrusive than Xprobe (on average). The significant improvement between aosd and hosd is again due to the combination of the passive and active modules (hosd needs to send fewer packets as it already possesses a lot of information from passive monitoring).

7.3.2 Group Candidate Query

We now look at the performance of OSD tools to filter non-critical IDS alarms, this experiment was initially presented in Section 3.2.4. The idea is to see how good are

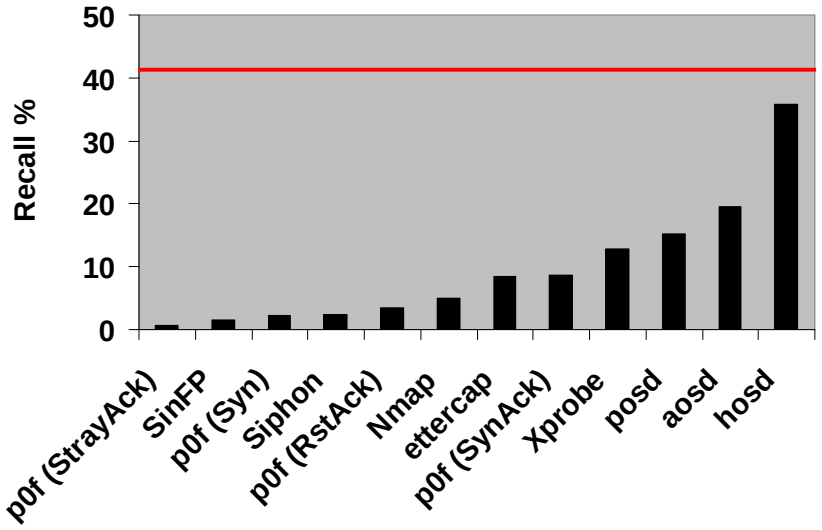


Figure 7.5: Recall for the Group Candidate Query

OSD tools in checking whether the target is vulnerable to a given attack. The results are presented in terms of precision and recall. The recall measure represents the amount of non-critical alarms filtered based on the information provided by an OSD tool; while precision indicates the amount of critical alarms that where erroneously filtered. We also consider the traffic generated by active tools, to compare against the results for the Exact Candidate Query.

7.3.2.1 Recall

Figure 7.5 presents the recall for posd, aosd, and hosd (with other OSD tools for comparison). The following observations are relevant:

- posd is better than any other passive tool. This again confirms our intuition that knowledge is the key for passive OSD.

- aosd is better than any other active tool. This is mainly due to the fact that active tools are typically limited to a handful of tests, to avoid flooding the network. By using careful test selection in aosd, we were able to include several tests, thus enhancing its accuracy without injecting more packets in the network (this will be shown in Section 7.3.2.3).

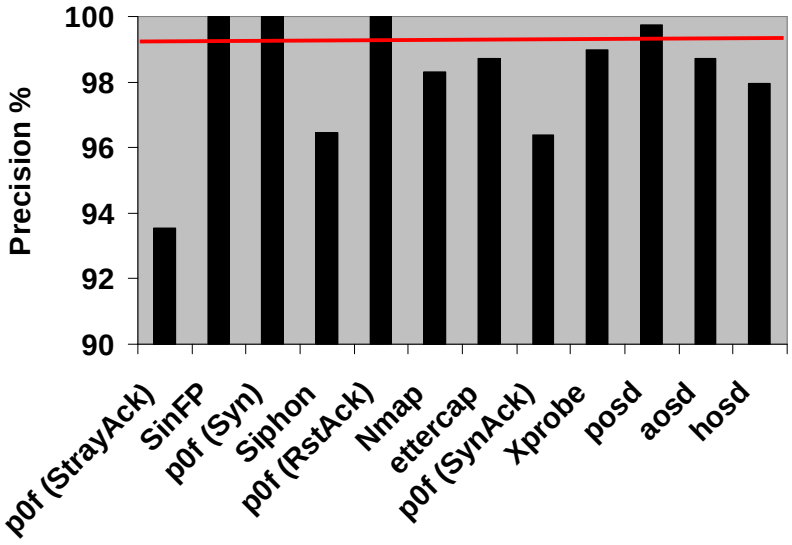


Figure 7.6: Precision for the Group Candidate Query

- The key observation here is that hosd, which combines both aosd and posd, is a significant improvement over both modules.

The overall improvement over state of the art OSD tools is clear. hosd is quite close to the optimum; which is achieved when knowing the exact configuration of each target (represented by the horizontal line in Figure 7.5).

7.3.2.2 Precision

Figure 7.6 compares the precision of our tools with other existing OSD tools. Here are the key observations:

- While the precision of posd is a little higher than most other tools, the one of hosd is a bit lower.

- The precision of hosd is below the lower bound (achieved when we know the actual target OS). This means that some extra mistakes are due to wrong guesses by hosd. Although the precision decrease is not dramatic, this definitely indicates room for improvement. It seems that the loss of precision is mainly due to the active module. The discrepancy between aosd and hosd is possibly due to the fact that different tests turned out to be executed when running the active

Table 7.3: Packet Injection Summary for Group Candidate Query

	Nb packets sent		
Tool	***Min***	***Mean***	***Max***
Nmap	882	1686	2186
Xprobe	7	7	7
aosd	1	7.9	16
hosd	0	2.1	8.4

module alone versus when running the active module on top of the passive one. However, further investigation is required.

- Note that some OSD tools, SinFP and p0f in both the Syn and SynAck modes, show an artificially high precision. It is artificial because these three tools have a very low recall rate; it is easy to have a perfect precision by never classifying any alarms as non-critical, but the downside is a null recall.

7.3.2.3 Traffic Generated

We observe, again, the number of packets injected by active tools (including aosd and hosd), but now when trying to identify if the actual OS is part of a specific group. Table 7.3 presents a summary of the number of packets injected by active tools. We observe the following:

- There is no difference for Nmap and Xprobe between the Group Candidate Query (presented here) and the Exact Candidate Query (see Section 7.3.1.3). This was expected since classical active OSD tools are not query-dependant. That is, they will send all their tests regardless of what the user wants to know.

- On the other hand, there is a difference for our tools, aosd and hosd, between the two queries. The Group Candidate Query requires significantly fewer tests to be executed; both on average and in the worst case. This confirms our intuition that a query based approach is valuable to lower the overall cost of active testing. We believe this claim to go beyond OSD and to apply to diagnosis in general.

7.4 Discussion

Experimental results confirm that an OSD tool can greatly benefit from adequate engineering.

From the passive point of view, a strong knowledge representation scheme provides a significant improvement. It enables the use of a memory (e.g., to remember previous deductions) and allows the tool to model complex phenomena requiring multiple packets (e.g., stimulus-response and tendency analysis).

On the active side, reasoning for test selection is important. This allows to have a wide variety of tests available, while avoiding to flood the network at each run (because only a handful of tests are executed in a given situation). Reasoning also gives the possibility to rely on different test selection strategies, where some could provide better, i.e., less intrusive solutions, while being more expensive (computationally speaking) and vice versa. Our experiments showed that the naive greedy approach is optimal in our specific situation (and it is computationally cheap). However, we cannot conclude that this will always be the case, even for OS discovery, as a different layout of the tests could break this property.

Yet, the most important improvement is definitely the combination of the passive and active approaches into a hybrid tool. A maximum number of phenomena can be considered by the a hybrid tool; some phenomena are observable only actively as they should never occur on a normal network, while some others cannot be triggered on demand and thus can only be observed passively.

7.4.1 Deployment

Deploying an OSD tool in a network is not a trivial task. Active tools are not a problem as they can be located anywhere inside the network. They will work as long as they can communicate with the fingerprinted hosts.

Passive tools, on the other hand, must be deployed in a strategic location to maximize the amount of traffic they see (e.g., ARP requests do not travel outside the network segment). For instance, deploying a passive OSD tool at the gateway guarantees to see communications between a local host and the internet, but it might not allow to see the communications between two hosts inside the network (e.g.,

between a workstation and a local data server). Most passive tools assume they are deployed in a central location with access to all the traffic.

Because it has a passive module, hosd should be deployed in a central location. However, since hosd can rely on its active module in case of insufficient information, it is not dramatic if it does not see all the traffic. The more traffic it sees, the less intrusive it should be, thus, it should be deployed in the most central location.

A better deployment strategy for passive OSD tool (and thus for hosd) would be to use a sensor-based architecture where we collect (and inject for hosd) traffic from several different location in the network. This would maximize the amount of traffic seen by the tool. The idea of extending hosd to support a sensor-based deployment is left as future work.

Chapter 8

Conclusion

Each success only buys an admission ticket to a more difficult problem.
-Henry Kissinger

This chapter provides an overview of the thesis. It first summarizes the work and provides the main conclusions obtained so far. It then presents the contributions and discusses some interesting ideas regarding possible areas for future work.

8.1 Thesis Summary

First, Chapter 2 introduced the problem of OS discovery and studied the classical approaches, active and passive, to OSD. Among other things, it exposed the shortcomings of existing tools:

- The lack of knowledge representation within passive tools.
- The lack of reasoning within active tools.

Then, Chapter 3 provided the motivations for our work on OS discovery. It discussed why OS discovery is important: mainly for identifying non-critical alarms in IDS context where information about the target OS can be used to identify more than 40% of non-critical alarms, assuming we know the actual target OS. It also showed that current tools are not adequate for gathering the required information, achieving only 1/3 of their potential.

Chapter 4 provided a background on the theory of diagnosis problem which plays an important role in this thesis. The principles of diagnosis can be applied to the OSD problem, providing a knowledge-oriented framework together with complexity results and well-studied algorithms.

Chapter 5 stated the problem addressed in this thesis, i.e., to develop a new OS discovery tool, and introduced our three main objectives:

- To develop a better OSD tool.
- To design the tool following a strong theoretical background.
- To provide a systematic (and automated) way of collecting OS fingerprints and incorporating them in our tool.

Based on that, Chapter 6 presented our contributions with respect to the three objectives mentioned above. It first presented the concept of hybrid OS discovery and showed how the OSD problem is nicely modeled as a diagnosis task. Then, it provided the necessary extensions to the theory of diagnosis, to allow queries on the knowledge base. Finally, it presented our work on virtual networks which can be used to automatically gather OS fingerprints at a low cost.

Finally, Chapter 7 discussed the implementation of our hybrid approach to OSD and presented experimental results comparing our tool with other OSD tools.

8.2 Conclusion

Based on the work presented in this thesis, we believe the following key points are worth mentioning:

- Target configuration information (i.e., operating system and application) is extremely relevant for the context of an attack. It can filter out a significant amount of non-critical alarms (40% when considering only the target OS, but up to 75% when considering the target applications as well [25]).
- Current OSD tools, however, are not adequate for the task of IDS context gathering; achieving only 1/3 of their potential. The following reasons explain why current tools are not adequate for context gathering:
  - Passive tools do not memorize past events nor previous deductions.
  - Passive tools do not consider multi-packet events.
  - Active tools are intrusive due to their lack of reasoning. Hence, they limit themselves to a very small amount of tests.

  - No tool provides the ability to continuously monitor the network.

- A knowledge-oriented approach to OS discovery greatly improves the accuracy; starting with passive OSD.

- Combining the active and passive approaches into a hybrid one increases the accuracy (by maximizing the number of phenomena that can be observed) while reducing the number of executed tests (by relying on passive information).

- Diagnosis theory is a very natural and useful formalization of the OS discovery problem. It provided an intuitive framework to reason about the problem. Moreover, it supplied a polynomial algorithm for the passive module.

- Extending diagnosis theory with a query-based approach generally reduces the cost of extracting information, as some queries are easier to solve than others.

- The fact that the greedy test selection strategy can be arbitrarily bad contradicts the claim that a one-step lookahead greedy strategy is good enough for test selection. However, it appears that for the specific test definitions found in our OS discovery instance, the greedy approach performs optimally.

8.3 Contributions

The results we obtained lead to contributions across different fields:

- Establishing the effectiveness of target configuration information for IDS context gathering [18, 24, 25, 50, 51]. Using target configuration information, we can identify around 75% of non-critical alarms and more than 40% when considering only the target OS.

- Proposing a hybrid approach to operating system discovery based on solid theoretical grounds [15, 21, 22, 23], i.e., the theory of diagnosis. This new approach provides a strong knowledge-representation component and combines the active and passive approaches together. As a result, the hybrid approach is more accurate and less intrusive than the classical approaches while being better suited for the IDS context gathering task.

- Extending the theory diagnosis with a query-based approach to give more flexibility to the framework and laying the foundations for a theory of test selection in diagnosis [19, 20]. These are the first steps toward minimizing the number of tests required to perform diagnosis.

- Designing a framework to automatically execute network experiments in a virtual environment [16, 17]. It can be used to automate the tedious process of gathering OS fingerprinting; lowering the costs, both in terms of time and hardware. But, our framework can also be used for other types of network experiments.

8.3.1 Tools

We have developed two open source tools:

- hosd. A hybrid tool for operating system discovery, available at `hosd.sourceforge.net`.

- VNEC. A tool to specify and execute network experiments in a virtual environment, available at `vnec.sourceforge.net`.

8.4 Future Work

The work presented in this thesis can be extended in many interesting ways. Possible areas for future work directly related to our OSD tool are:

- Adopting a multiple faults diagnosis model [31] to handle network address translators (NATs). This model, however, is known to be computationally more expensive; it requires replacing the algorithm presented in Figure 6.3 with a more general procedure to find the minimum hitting sets, which is known to be NP-Complete [59]. It is worth noting, however, that existing OSD tools do not handle NATs either.

- Relying on prior probabilities to take into account the popularity of different OSes. Indeed, even without any observations, a computer is more likely to be

running Windows XP sp3 than FreeBSD 6.3. These *prior probabilities* can be incorporated into the diagnosis model, provided we modify the candidate generation algorithm to update the probability of each candidate, see [28]. Moreover, those probabilities might be useful to guide the test selection process[1], e.g., replacing the entropy-based measure presented in Section 6.2.2 by a maximum expected utility measure [67]. One difficulty will be to obtain good estimates about the popularity of different OSes, not only at the family level (e.g., Windows vs Linux), but also at the release and version levels (e.g., XP vs vista and sp2 vs sp3).

- In this thesis we assumed the existence of a central network point where to install the OSD tool for traffic collection and analysis. It would be better to deploy multiple instances of the OSD tool (sensors) in different strategic locations across the network. Each instance is responsible for analyzing, in a completely independent way, the traffic it receives and for maintaining its own partial representation. Then, a central unit is deployed to handle the queries made by the user. The central unit requests, from the sensors, information relevant to the query, and merge it to form a detailed representation. A simplified version of the distributed diagnosis theory of [34] would provide the necessary extension to our current diagnosis model.

Other interesting areas for future work are:

- Concerning the use of contextual information to eliminate non-critical alarms in IDSes, an experiment comparing the different contextual approaches (gathering target configuration information, running vulnerability assessment tools, analyzing attack side effects) could be run. The goal is to see if the different approaches can complement each other.

- It would be interesting to generalize the test selection theory to other diagnosis domains, i.e., with different properties, constraints, and assumptions than OSD.

- With respect to our virtual environment VNEC, the following extensions are under way to make the tool more suitable for generic network experiments:

[1] They might be used as a heuristic to guide the ordering of the tests.

- Extend the architecture to support the distribution of an experiment across multiple physical machines. This will allow to run experiments requiring larger networks.
- Extend the architecture to support multiple virtualization technologies. The objective is to be able to build a network with VMs from different technologies (e.g., VMWare, VirtualPC, VirtualBox, etc.) in such a way that the VMs can still communicate with one another. This will give us access to a wider variety of OSes.

Bibliography

[1] Jon M. Allen. OS and Application Fingerprinting Techniques. SANS Institute InfoSec Reading Room, 2007.

[2] Annie De Montigny-Leboeuf. A Multi-Packet Signature Approach to Passive Operating System Detection. Technical Report CRC-TN-2005-001, Communications Research Centre Canada, January 2005.

[3] Ofir Arkin and Fyodor Yarochkin. Xprobe Homepage. `http://xprobe.sourceforge.net`.

[4] Mikhail Atallah. *Algorithms and Theory of Computation Handbook*. Applied Algorithms and Data Structures series. CRC-Press, 1st edition edition, 1998.

[5] Patrice Auffret. SinFP Homepage. `http://www.gomor.org/cgi-bin/sinfp.pl`.

[6] Reuven Bar-Yeuda and Zahavit Kehat. Approximating the Dense Set-Cover Problem. *Journal of Computer and System Sciences*, 69:547–561, 2004.

[7] Chitta Baral, Sheila McIlraith, and Tran Cao Son. Formulating Diagnostic Problem Solving using an Action Language with Narratives and Sensing. *Proceedings of the 7th conference on Principles of Knowledge Representation and Reasoning (KR'00)*, pages 311–322, 2000.

[8] Mihir Bellare, Shafi Goldwasser, Carsten Lund, and A. Russeli. Efficient Probabilistically Checkable Proofs and Applications to Approximation. *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC'93)*, pages 294–304, 1993.

[9] Piotr Berman and Feorg Schnitger. On the Complexity of Approximating the Independent Set Problem. *Information and Computation*, 96(1):77–94, 1992.

[10] Piergiorgio Bertoli, Allessandro Cimatti, John Slaney, and Sylvie Thiébaux. Solving Power Supply Restoration Problems with Planning via Symbolic Model-Checking. *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02)*, pages 576–580, 2002.

[11] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithms*. Prentice Hall, 1996.

[12] Patrick Charles. jpcap homepage. `http://jpcap.sourceforge.net/`.

[13] Gary Chartrand and Linda Lesniak. *Graphs & Digraphs*. Chapman & Hall, 2004.

[14] François Gagnon. VNEC Homepage. `http://vnec.sourceforge.net`.

[15] François Gagnon. Operating System Discovery Using Answer Set Programming. ACAI 2007 summer school - Poster Session, 2007.

[16] François Gagnon, Tomas Dej, and Babak Esfandiari. VNEC - A Virtual Network Experiment Controller. *Proceedings of the 2nd International Workshop on Systems and Virtualization Management (SVM'08)*, pages 119–124, 2008.

[17] François Gagnon, Tomas Dej, and Babak Esfandiari. Network in a Box. *(Submitted to) 2010 International Conference on Data Communication Networking (DCNET'10)*, 2010.

[18] François Gagnon and Babak Esfandiari. *Emerging Artificial Intelligence Applications in Computer Engineering*, volume 160 of *Frontiers in Artificial Intelligence and Applications*, chapter Using Artificial Intelligence for Intrusion Detection, pages 295–306. IOS Press, 2007.

[19] François Gagnon and Babak Esfandiari. A Query-Based Approach for Test Selection in Diagnosis - Operating System Discovery as a Case Study. *Proceedings of the 19th International Workshop on Principles of Diagnosis - Poster Session (DX'08)*, 2008.

[20] François Gagnon and Babak Esfandiari. A query-based approach for test selection in diagnosis. *Artificial Intelligence Review*, 29(3):249–263, 2009.

[21] François Gagnon and Babak Esfandiari. Using Answer Set Programming to Enhance Operating System Discovery. *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 579–584, 2009.

[22] François Gagnon and Babak Esfandiari. A hybrid approach to operating system discovery based on diagnosis. *(Accepted for publication in) International Journal of Network Management*, 2010.

[23] François Gagnon, Babak Esfandiari, and Leopoldo Bertossi. A Hybrid Approach to Operating System Discovery Using Answer Set Programming. *Proceedings of the 10th IFIP/IEEE Symposium on Integrated Management (IM'07)*, pages 391–400, 2007.

[24] François Gagnon, Frédéric Massicotte, and Babak Esfandiari. On the Effectiveness of Target Configuration as Contextual Information for IDS Alarm Classification. Technical Report SCE-08-08, Department of Systems and Computer Engineering - Carleton University, 2008. `http://www.sce.carleton.ca/~fgagnon/Publications/context.pdf`.

[25] François Gagnon, Frédéric Massicotte, and Babak Esfandiari. Using Contextual Information for IDS Alarm Classification. *Proceedings of the 6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'09)*, pages 147–156, 2009.

[26] Luca Console and Pietro Torasso. A Spectrum of Logical Definitions of Model-Based Diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[27] Pierluigi Crescenzi. A Short Guide to Approximation Preserving Reductions. *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity (CCC'97)*, pages 262–273, 1997.

[28] Johan de Kleer. *Readings in Model-Based Diagnosis*, chapter Focusing on Probable Diagnoses, pages 131–137. Morgan Kaufmann, 1992.

[29] Johan de Kleer. *Readings in Model-Based Diagnosis*, chapter Using Crude Probability Estimates to Guide Diagnosis, pages 118–124. Morgan Kaufmann, 1992.

[30] Johan de Kleer, Olivier Raiman, and Mark Shirley. *Readings in model-based diagnosis*, chapter One Step Lookahead is Pretty Good, pages 138–142. Morgan Kaufmann Publishers, 1992.

[31] Johan de Kleer and Brian C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[32] Johan de Kleer and Brian C. Williams. Diagnosis with Behavrioral Modes. *Proceedings of the 1989 International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 1324–1330, 1989.

[33] Irit Dinur and Samuel Safra. On the Hardness of Approximating Minimum Vertex Cover. *Annals of Mathematics*, 162:439–485, 2005.

[34] Eric Fabre, Albert Benveniste, and Claude Jard. Distributed Diagnosis for Large Discrete Event Dynamic Systems. *Proceedings of the 15th IFAC World Congress on Automatic Control*, 2002.

[35] Uriel Feige. A Threshold of $ln\ n$ for Approximating Set Cover. *Journal of the ACM*, 45(4):634–652, July 1998.

[36] Foundstone. OS Identification Methods and Countermeasures. Foundstone Strategic Security white paper, August 2003.

[37] Gerhard Friedrich and Wolfgang Nejdl. Choosing Observations and Actions in Model-Based Diagnosis/Repair Systems. *Proceedings of the International Conference on Knowledge Representation and Reasoning*, pages 489–498, 1992.

[38] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman and Company, 1979.

[39] Michael R. Genesereth. The Use of Design Descriptions in Automated Diagnosis. *Artificial Intelligence - Special volume on qualitative reasoning about physical systems*, 24(1-3):411–436, 1984.

[40] Eran Halperin. Improved Approximation Algorithms for the Vertex Cover Problem in Graphs and Hypergraphs. *SIAM Journal on Computing*, 31(5):1608–1623, 2002.

[41] Dorit S. Hochbaum. Approximation Algorithms for the Set Covering and Vertex Cover Problems. *SIAM Journal on Computing*, 11(3):555–556, 1982.

[42] Dorit S Hochbaum. *Approximation Algorithm fo HP-Hard Problems*. PWS Publisher Company, 1997.

[43] S. Impedovo, L. Ottaviano, and S. Occhinegro. Optical character recognition - a survey. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 5(1/2):1–24, June 1991.

[44] David S. Johnson. Approximation Algorithms for Combinatorial Problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[45] Gary Kessler. *Handbook on Local Area Networks*, chapter 24 - IPv6: The Next Generation Internet Protocol, pages 277–292. Auerbach Publications, 1999.

[46] Gary Kessler. An Overview of TCP/IP Protocols and the Internet. InterNIC, 2007.

[47] Eric Kollmann. Chatter on the Wire: A look at excessive network traffic and what it can mean to network security, 2005.

[48] Carsten Lund and Mihalis Yannakakis. On the Hardness of Approximating Minimization Problems. *Journal of the ACM*, 41(5):960–981, 1994.

[49] Anthony T. Mann. *The Rational Guide To: Microsoft Virtual PC 2004*. Rational Guides. Rational Press, 2004.

[50] Frédéric Massicotte and François Gagnon. A Publicly Available Data Set for the Evaluation of Signature-Based IDS. poster session of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06), 2006.

[51] Frédéric Massicotte, François Gagnon, Mathieu Couture, Yvan Labiche, and Lionel Briand. Automatic Evaluation of Intrusion Detection Systems. *Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[52] Frédéric Massicotte, Mathieu Couture, and Annie De Montigny-Leboeuf. Using a VMware Network Infrastructure to Collect Traffic Traces for Intrusion Detection Evaluation. *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.

[53] Frédéric Massicotte, Tara Whalen, and Claude Bilodeau. Network Mapping Tool for Real-Time Security Analysis. *NATO/RTO Information Systems Technology Panel Symposium on Real Time Intrusiton Detection*, 2002.

[54] Sheila McIlraith. Generating Tests using Abduction. *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 449–460, 1994.

[55] Sheila McIlraith. Explanatory Diagnosis: Conjecturing Actions to Explain Observations. *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 167–177, 1998.

[56] Sheila McIlraith and Richard Scherl. What Sensing Tells Us: Towards a Formal Theory of Testing for Dynamical Systems. *Proceedings of the National Conference on Artificial Intelligence (AAAI'00)*, pages 483–490, 2000.

[57] Jose Nazario. Passive System Fingerprinting using Network Client Applications. Crimelabs Research Report, November 2000.

[58] Alberto Ornaghi and Marco Valleri. Ettercap Homepage. `http://ettercap.sourceforge.net`.

[59] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[60] Ramesh Patil, Peter Szolovits, and William Schwartz. Causal Understanding of Patient Illness in Medical Diagnosis. *Proceedings of International Joint Conference on Aritificial Intelligence (IJCAI'81)*, 2:893–899, 1981.

[61] David Poole. Normality and Faults in Logic-Based Diagnosis. *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI'85)*, pages 1304–1310, 1985.

[62] David Poole. Representing Knowledge for Logic-Based Diagnosis. *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1282–1290, 1988.

[63] David Poole. Representing Diagnosis Knowledge. *Annals of Mathematics and Artificial Intelligence*, 11(1-4):33–50, March 1994.

[64] David Poole, Randy Goebel, and Romas Aleliunas. *The Knowledge Frontier - Essays in the Representation of Knowledge*, chapter 13 - Theorist: A Logical Reasoning System for Defaults and Diagnosis, pages 331–352. Symbolic Computation. Springer-Verlag, 1987.

[65] Antonia Rana. What is AMap and how does it fingerprint applications? SANS Institute.

[66] Raymond Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[67] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[68] SecurityFocus. SecurityFocus Homepage. `http://www.securityfocus.org/`.

[69] Amit Singhal. Modern Information Retrieval: A Brief Overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35–43, 2001.

[70] Matthew Smart, Robert Malan, and Farnam Jahanian. Defeating TCP/IP Stack Fingerprinting. *Proceedings of the 9th USENIX Security Symposium*, pages 229–240, 2000.

[71] Peter Struss. Testing for Discrimination of Diagnoses. *In Proceedings of the National Conference on Artificial Intelligence (AAAI'94)*, pages 251–256, 1994.

[72] Subterrain Security Group. Siphon Homepage. `http://siphon.datanerds.net/`.

[73] Sylvie Thiébaux, Marie-Odile Cordier, Olivier Jehl, and Jean-Paul Krivine. Supply restoration in Power Distribution Systems - A Case Study in Integrating Model-Based Diagnosis and Repair Planning. *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI'96)*, pages 525–532, 1996.

[74] Chris Trowbridge. An Overview of Remote Operating System Fingerprinting. *SANS InfoSec Reading Room - Penetration Testing*, October 2003.

[75] VMWare. VMWare Homepage. `http://www.vmware.com/`.

[76] Steven Warren. *The VMWare Workstation 5 Handbook (Networking and Security)*. Charles River Media, 2005.

[77] Andtrew Whitaker and Daniel Newman. *Penetration Testing and Network Defense*. Cisco Press, 2006.

[78] Lars Wirzenius, Joanna Oja, Stephen Stafford, and Stephen Stafford. Linux System Administrators Guide. http://tldp.org/LDP/sag/html/index.html, 2003.

[79] Fyodor Yarochkin. Nmap Homepage. http://www.insecure.org/nmap/.

[80] Fyodor Yarochkin. Remote OS Detection via TCP/IP Stack FingerPrinting. http://www.insecure.org/nmap/nmap-fingerprinting-article.html, October 1998.

[81] Michal Zalewski. p0f homepage. http://lcamtuf.coredump.cx/p0f.shtml.

Appendix A

OSD Tests

A.1 Test Descriptions

Definition A.1 (Test-1 (TCP Syn))
This test considers the first packet of a TCP handshake and will provide us with information on the sender of the packet. Example of interesting information extracted from that test are:

- *Is the DF bit set ?*
- *What is the value of the TTL field ?*
- *What is the value of the WIN field ?*
- *What are the TCP options advertised ?*

This test is single-packet, standard, and exclusively passive. ◯

Definition A.2 (Test-2 (ARP Request))
This test focuses on the ARP requests made by a computer and will provide information about the sender. The field of interest here is the target hardware address (which is unknown to the sender) and can take any value. This test is single-packet, standard, and can be used both passively and actively[1]*.* ◯

Definition A.3 (Test-3 (TCP ISN))
This test analyzes the generation algorithm for the initial sequence number of a TCP handshake negotiation. Most OSes increment the ISN by a random number each time, while others increment it by a constant value. This test requires a sample of packets, is standard, and is exclusively passive. ◯

[1] By forging a SYN packet where the sender IP is an unused address.

Definition A.4 (Test-4 (IP ID))

Test-4 looks at the unique identifier of consecutive IP packets to extract the generation pattern of that value. Most OSes will use a constant increment of one, while others might use a different constant value or even a random value. This test also requires a sample of packets, is standard, and is both active and passive. ◯

Definition A.5 (Test-5 (TCP TS))

This test studies the timestamp refresh rate as advertised in the TCP options. By sending a SYN packet with the timestamp TCP options, we obtain a SYN/ACK packet with a timestamp value (if the target OS supports the timestamp option). Sending several such stimuli can provide us with the target update rate of the timestamp value. Some OSes will update the value twice per seconds while others will update it 1000 times per seconds. This test is based on a sample of packets, is standard, and is both active and passive. ◯

Definition A.6 (Test-6 (ARP Retransmit))

This test observes the number of times an unanswered ARP request is retransmitted and the delays between each retransmission. This is a sample test which is standard and can be used both actively and passively. ◯

Definition A.7 (Test-7 (ICMP ID SEQ))

Similar to the IP ID test (see Test-5 in Definition A.5), this test considers the unique identifier generation algorithm for ICMP packets (as well as the ICMP sequence number generation). This test requires a sample of packet, is standard, and passive. ◯

Definition A.8 (Test-8 (SynAck))

This test analyzes how the target behaves during the second step (SYN/ACK) of the TCP handshake (when it receives a SYN request on an open port). This test considers fields such as DF, TTL, WIN, TCP options, etc. This test is of type stimulus-response, but could also be handled as a single-packet test (with some loss of information). It is standard and can be active as well as passive. ◯

Definition A.9 (Test-9 (RstAck))

This test studies how a computer reacts (RST/ACK) to a SYN request on a closed port. The fields considered are similar to those of the SynAck test (see Test-8 in

Definition A.8). This test is stimulus-response, it is standard, and both active and passive. ○

Definition A.10 (Test-10 (ICMP Unreach))
This test analyzes the way a computer responds (ICMP port unreachable) to a UDP packet sent on a closed UDP port. This test considers, among others, the fields DF, TTL, TOS (Type Of Service). This is a stimulus-response, it is standard and both active and passive. ○

Definition A.11 (Test-11 (ICMP Echo))
This test sends an ICMP Echo request and studies the corresponding ICMP Echo reply. The fields of interests are: DF, TTL, TOS, and the ICMP code in the reply. This test is a stimulus-response, it is standard and can be used both actively and passively. ○

Definition A.12 (Test-12 (ICMP Info))
This test sends an ICMP Info request and studies the corresponding ICMP Info reply. The fields of interests are: DF, TTL, TOS, and the ICMP code in the reply. This test is a stimulus-response test. It is not clear whether this test should be considered standard or not; it was once part of a protocol standard which is now obsolete, see Section 4.3.3.7 of RFC 1812). This test can be used both actively and passively (although we should not expect to see this kind of messages naturally on the network). ○

Definition A.13 (Test-13 (ICMP TS))
Exactly like Test-11 (see Definition A.11) except that it relies on the pair ICMP Timestamp request/reply. ○

Definition A.14 (Test-14 (ICMP Mask))
Exactly like Test-11 (see Definition A.11) except that it relies on the pair ICMP Mask request/reply. ○

Definition A.15 (Test-20 (SynEcn))
This test studies how a computer responds to a TCP packet with the flags Syn and Ecn set sent to an open port. Most OSes will answer with a SYN/ACK packet; however, some OSes will not respond while others will respond with a SYN/ACK/ECN packet.

The fields of interest are: flags, DF, TTL, WIN, TCP options. This test is stimulus-response, standard, and active only. ◯

Definition A.16 (Test-21 (no flag))

This test analyzes how an OS responds to a TCP packet having no flag set and sent to an open port. Some OSes will reply with a RST/ACK packet, but others will fail to reply. Fields of interest are like Test-20. This is a stimulus-response test, it is non-standard (a TCP packet should always have some flags set), and active only. ◯

Definition A.17 (Test-22 (SynFinUrgPsh))

This test studies how an OS responds to a TCP packet with the flags Syn, Fin, Urg, and Psh set. Most OSes will reply with a SYN/ACK packet, while some will reply with a SYN/ACK/FIN packet and others will not respond. The fields of interest are like in Test-20. This is again a stimulus-response test, it is also non-standard (Syn and Fin flags cannot be set together), and is only active. ◯

Definition A.18 (Test-23 (Ack open))

This test examines how an OS responds to a TCP packet having only the Ack flag set and sent to an open port when no TCP connection has been established beforehand. The fields of interest are like Test-20. This test is stimulus-response, standard and active only. ◯

Definition A.19 (Test-24 (Ack closed))

Exactly like Test-23, except that the stimulus is now sent on a closed port. ◯

Definition A.20 (Test-25 (FinUrgPsh))

This test examines how an OS responds to a TCP packet sent to a closed port with flags Fin, Urg, and Psh set. The fields of interest are like Test-20. This test is stimulus-response, standard and active only. ◯

Definition A.21 (Test-26 (Echo Request))

This test examines the content of the ICMP Echo request packet sent by the computer. Three fields are of interest here: DF, TTL and padding[2]*. For instance, Windows uses 32 bits while Linux uses 56 bits of padding data for echo request packets.* ◯

[2]Padding is the insertion of meaningless data to ensure a packet respect the size requirements: Ethernet packets must have a minimum size of 512 bits and the total size must be divisible by 32

Table A.1: Fingerprints for the RstAck Tests

Fingerprint ID	DF	TTL	WIN
1	no	255	0
2	no	64	0
3	yes	255	0
4	yes	64	0
5	yes	Echoed	0
6	Echoed	255	0
7	Echoed	64	0
8	no	128	0
9	yes	128	0
10	no	64	Echoed
11	no	32	0

A.2 Test Results

Here we provide an example of how OS behavior can be used to partition the space of OSes and how the result of a test can be used to discard some OSes. We use the RstAck test (see Definition A.9) and we consider the DF, TTL and WIN field of the RST/ACK packet generated by the target. DF takes values yes or no, TTL takes any value between 1 and 255 (usually a power of 2) and WIN takes any value between 0 and 65,535. Note that the WIN field is not meaningful in a RST/ACK packet, so OSes can fill it as they want. Table A.1 shows the 11 different fingerprints we have observed so far for the RstAck test ("Echoed" means the value in the RST/ACK packet is the same as the value in the SYN packet). Based on this table, we see that the RstAck test partitions the set of OSes into 11 classes.

Table A.2 associates 211 OSes with its corresponding fingerprint. From there, we see that if a machine sends a RST/ACK packet with DF=no, TTL=32, and WIN=0, then it is either Windows 95 or Windows NT 3.51 workstation (as provided by fingerprint 11). So this event provides a lot of information as it discards 209 out of the 211 OSes considered. On the other hand, if a machine sends a RST/ACK packet with DF=no, TTL=64, and WIN=0, then it runs one of the 87 OSes associated with fingerprint 2.

Table A.2: OS Fingerprint Associations for Test RstAck

Fingerprint ID	**Operating System**
1	BEOS 5
1	FreeBSD 4.10
1	FreeBSD 4.11
1	FreeBSD 4.9
1	Linux 2.2.0
1	Linux 2.2.1
1	Linux 2.2.10
1	Linux 2.2.11
1	Linux 2.2.12
1	Linux 2.2.12-20
1	Linux 2.2.13
1	Linux 2.2.14
1	Linux 2.2.14-5
1	Linux 2.2.15
1	Linux 2.2.16
1	Linux 2.2.16-22
1	Linux 2.2.17
1	Linux 2.2.18
1	Linux 2.2.2
1	Linux 2.2.20
1	Linux 2.2.20-idepci
1	Linux 2.2.21
1	Linux 2.2.22
1	Linux 2.2.23
1	Linux 2.2.24
1	Linux 2.2.3
1	Linux 2.2.4

Continued on next page

Table A.2 – OS Fingerprint Associations - continued from previous page

Fingerprint ID	**Operating System**
1	Linux 2.2.5
1	Linux 2.2.5-15
1	Linux 2.2.6
1	Linux 2.2.7
1	Linux 2.2.8
1	Linux 2.2.9
1	NetBSD 1.6.2
2	FreeBSD 2.0.5
2	FreeBSD 2.1.0
2	FreeBSD 2.1.5
2	FreeBSD 2.1.6
2	FreeBSD 2.1.7.1
2	FreeBSD 2.2.0
2	FreeBSD 2.2.1
2	FreeBSD 2.2.2
2	FreeBSD 2.2.5
2	FreeBSD 2.2.6
2	FreeBSD 2.2.7
2	FreeBSD 2.2.8
2	FreeBSD 3.0
2	FreeBSD 3.1
2	FreeBSD 3.2
2	FreeBSD 3.3
2	FreeBSD 3.4
2	FreeBSD 3.5.1
2	FreeBSD 4.0
2	FreeBSD 4.1
2	FreeBSD 4.1.1

Continued on next page

Table A.2 – OS Fingerprint Associations - continued from previous page

Fingerprint ID	**Operating System**
2	FreeBSD 4.2
2	FreeBSD 4.3
2	FreeBSD 4.4
2	FreeBSD 4.5
2	FreeBSD 4.6
2	FreeBSD 4.6.2
2	FreeBSD 4.7
2	FreeBSD 4.8
2	FreeBSD 5.0
2	FreeBSD 5.1
2	FreeBSD 5.2
2	FreeBSD 5.2.1
2	FreeBSD 5.3
2	FreeBSD 5.4
2	Linux FC1 2.4.22-1.2115.nptl
2	Linux FC2 2.6.5-1.358
2	Linux FC3 2.6.9-1.667
2	Linux FC4 2.6.11-1.1369_FC4
2	Linux SUSE10
2	Linux SUSE82 2.4.20-4GB
2	Linux SUSE90 2.4.21-99-defaulf
2	Linux SUSE91 2.6.4-52-default
2	Linux SUSE92 2.6.8-24-default
2	Linux SUSE93 2.6.11.4-20a-default
2	MacOS 10 10.1.0 Workstation
2	MacOS 10 10.1.1 Workstation
2	MacOS 10 10.1.2 Workstation
2	MacOS 10 10.1.3 Workstation

Continued on next page

Table A.2 – OS Fingerprint Associations - continued from previous page

Fingerprint ID	**Operating System**
2	MacOS 10 10.1.4 Workstation
2	MacOS 10 10.1.5 Workstation
2	MacOS 10 10.2.1 Workstation
2	MacOS 10 10.2.2 Workstation
2	MacOS 10 10.2.3 Workstation
2	MacOS 10 10.2.4 Workstation
2	MacOS 10 10.2.5 Workstation
2	NetBSD 1.1
2	NetBSD 1.2
2	NetBSD 1.2.1
2	NetBSD 1.3
2	NetBSD 1.3.1
2	NetBSD 1.3.2
2	NetBSD 1.3.3
2	NetBSD 1.4
2	NetBSD 1.4.1
2	NetBSD 1.4.2
2	NetBSD 1.4.3
2	NetBSD 1.5
2	NetBSD 1.5.1
2	NetBSD 1.5.2
2	NetBSD 1.5.3
2	NetBSD 1.6
2	NetBSD 1.6.1
2	OpenBSD 2.0
2	OpenBSD 2.1
2	OpenBSD 2.2
2	OpenBSD 2.3

Continued on next page

Table A.2 – OS Fingerprint Associations - continued from previous page

Fingerprint ID	**Operating System**
2	OpenBSD 2.4
2	OpenBSD 2.5
2	OpenBSD 2.6
2	OpenBSD 2.7
2	OpenBSD 2.8
2	OpenBSD 3.4
2	OpenBSD 3.5
2	QNX RTP 6.1
2	QNX RTP 6.2
2	QNX RTP 6.2.1
3	Linux 2.4.0
3	Linux 2.4.1
3	Linux 2.4.10
3	Linux 2.4.10-4GB
3	Linux 2.4.11
3	Linux 2.4.12
3	Linux 2.4.13
3	Linux 2.4.14
3	Linux 2.4.15
3	Linux 2.4.16
3	Linux 2.4.17
3	Linux 2.4.18
3	Linux 2.4.18-3
3	Linux 2.4.18-4GB
3	Linux 2.4.2
3	Linux 2.4.2-2
3	Linux 2.4.3
3	Linux 2.4.4

Continued on next page

Table A.2 – OS Fingerprint Associations - continued from previous page

Fingerprint ID	**Operating System**
3	Linux 2.4.4-4GB
3	Linux 2.4.5
3	Linux 2.4.6
3	Linux 2.4.7
3	Linux 2.4.8
3	Linux 2.4.9
3	MacOS 9 9.1 Workstation
3	MacOS 9 9.2.1 Workstation
3	MacOS 9 9.2.2 Workstation
3	Windows 2000 sp1 Server
3	Windows 2000 sp1 Workstation
3	Windows 2000 sp2 Server
3	Windows 2000 sp3 Server
3	Windows 2000 sp4 Server
3	Windows 2000 std Server
3	Windows 2003 std sp1 Server
3	Windows NT 4 sp3 Server
3	Windows XP Home sp1a Workstation
3	Windows XP Home sp2 Workstation
3	Windows XP Professional sp1 Workstation
3	Windows XP Professional sp2 Workstation
4	Linux 2.4.18-14
4	Linux 2.4.19
4	Linux 2.4.19-4GB
4	Linux 2.4.20
4	Linux 2.4.20-8
4	Linux 2.4.21-0.13mdk
4	OpenBSD 2.9

Continued on next page

Table A.2 – OS Fingerprint Associations - continued from previous page

Fingerprint ID	**Operating System**
4	OpenBSD 3.0
4	OpenBSD 3.1
4	OpenBSD 3.2
4	OpenBSD 3.3
4	SunOS 5.8
4	SunOS 5.9
4	SunOS (Intel) 5.8
5	MacOS 7 7.5.3 Workstation
5	MacOS 7 7.5.5 Workstation
5	MacOS 7 7.6 Workstation
5	MacOS 7 7.6.1 Workstation
5	MacOS 8 8.0 Workstation
5	MacOS 8 8.1 Workstation
5	SunOS 5.5
5	SunOS 5.6
5	SunOS 5.7
6	MacOS 9 9.0 Workstation
7	Linux 2.2.16-SUSE
7	Linux 2.2.18-SUSE
7	MacOS 10 10.2.6 Workstation
8	Linux 2.4.7-RH
8	Netware 4.11 std
8	Windows 2000 sp2 Workstation
8	Windows 2000 sp3 Workstation
8	Windows 2000 sp4 Workstation
8	Windows 2000 std Workstation
8	Windows 2003 std Server
8	Windows 98 SE Workstation

Continued on next page

Table A.2 – OS Fingerprint Associations - continued from previous page

Fingerprint ID	**Operating System**
8	Windows 98 std Workstation
8	Windows Millenium std Workstation
8	Windows Net std Workstation
8	Windows XP Home Workstation
8	Windows XP Professional Workstation
9	Netware 4.11 sp9
9	Netware 5 sp6a
9	Netware 5 std
9	Netware 5.1 sp6
9	Netware 5.1 std
9	Netware 6 sp3
9	Netware 6 std
10	QNX RTP 4
10	QNX RTP 6.0
11	Windows 95
11	Windows NT 3.51 std Workstation

Appendix B

Evaluation Dataset

The dataset used for evaluation is taken from [51]. It consists of 6,656 traffic traces. Each trace contains the traffic generated by launching one of 92 exploits (see Table B.1) against one of the 95 targets available (see Table B.2). Each exploit was used against all the targets offering a service on the port targeted by the exploit. When an exploit allowed it, it was launched several times against the same target by using different parameters (e.g., the exploits can try different padding strings for buffer overflow). More information about the experiment environment can be found in [52].

When running an exploit against a specific target, a very simple network configuration is used. Only the attacker, the target, and sometimes extra network devices needed by the attack (e.g., DNS server) are up. They are all connected on the same segment and the traffic is recorded by the attacker.

Every host used during an attack (i.e., attacker, target, and network devices) are virtual machines (using VMWare Workstation technology [75]). To avoid the effects of a successful attack from being carried on to the next exploit execution, the targets are reverted to a clean state after each attack attempt.

Table B.1: Exploit List

Name	**Bugtraq ID**
0x333hate.c	7294
0x82-Remote.54AAb4.xpl.c	7294
0x82-WOOoou Happy_new.c	8315
0x82-dcomrpc_usemgret.c	8205
0x82-wu262.c	8315
30.07.03.dcom.c	8205
Continued on next page	

Table B.1 – Exploit List

Name	**Bugtraq ID**
7350oftpd.tar.gz	2124
ALL_UNIEXP.C	1806
DComExpl_UnixWin32.zip	8205
DDK-IIS.c	4485
HOD-ms04011-lsasrv-expl.c	10108
HOD-ms04031-expl.c	11372
IIS5.0_SSL.c	10115
IIS_escape_test.sh	2708
Iisenc.zip	2708
MS03-039-linux.c	8459
MS03-04.W2kFR.c	8826
MS04-007-dos.c	9635
MultiWinNuke.c	6005
RFParalyze.c	1163
THCIISSLame.c	10116
Xnuxer.c	7116
apache2.pl	2503
apache_chunked_win32.pm	5033
bid3581.txt	3581
bysin2.c	7230
crash_winlogon.c	1331
dcom.c	8205
decodecheck.pl	2708
decodexecute.pl	2708
execiis.c	2708
fpse2000ex.c	2906

Continued on next page

Table B.1 – Exploit List

Name	**Bugtraq ID**
ftpglob.nasl	3581
iis-zang.c	1806
iis40_htr.pm	307
iis50_printer_overflow.pm	2674
iis5hack.pl	2674
iis_nsiislog_post.pm	8035
iis_printer_bof.c	2674
iis_source_dumper.pm	1578
iis_w3who_overflow.pm	11820
iisex.c	2708
iisrules.pl	2708
iisrulessh.pl	2708
iisuni.c	1806
iiswebexplt.pl	2674
jill.c	2674
kod.c	514
kox.c	514
lala.c	2708
linux-wb.c	7116
lsass_ms04_011.pm	10108
m00-apache-w00t.c	3335
ms03-043.c	8826
ms05_039_pnp.pm	14513
msadc.pl	529
msasn1_ms04_007_killbill.pm	9633
msdtc_dos.nasl	4006

Continued on next page

Table B.1 – Exploit List

Name	**Bugtraq ID**
msftp_dos.pl	4482
msftp_fuzz.pl	4482
msrpc_dcom_ms03_026.pm	8205
mssql2000_preauthentication.pm	5411
mssql2000_resolution.pm	5311
oc192-dcom.c	8205
pimp.c	514
rfpoison.py	754
rpc!exec.c	8205
rs_iis.c	7116
samba_exp2.tar.gz	7294
samba_nttrans.pm	7106
samba_trans2open.pm	7294
sambal.c	7294
sambash.c	7106
servu_mdtm_overflow.pm	9751
smbnuke.c	5556
sol2k.c	2674
solaris_sadmind_exec.pm	8615
solaris_snmpxdmid.pm	2417
sslbomb.c	10115
unicodecheck.pl	1806
unicodexecute2.pl	1806
warftpd_165_pass.pm	10078
warftpd_165_user.pm	10078
wd.pl	7116

Continued on next page

Table B.1 – Exploit List

Name	Bugtraq ID
win_msrpc_lsass_ms04-11_Ex.c	10108
windows_ssl_pct.pm	10116
winnuke._eci.c	2010
winnuke.c	6005
winnuke.pl	2010
wins.c	11763
wins_ms04_045.pm	11763
zp-exp-telnetd.c	3064

Table B.2: Target List

OS	Applications
FreeBSD40	• 25/tcp - Sendmail Consortium Sendmail 8.3.9
FreeBSD410	• 137/udp - Samba Samba 2.2.8 a (Build 2) • 138/udp - Samba Samba 2.2.8 a (Build 2) • 139/tcp - Samba Samba 2.2.8 a (Build 2) • 25/tcp - Sendmail Consortium Sendmail 8.12.11 • 80/tcp - Apache Software Foundation Apache 1.3.29
FreeBSD4110	• 25/tcp - Sendmail Consortium Sendmail 8.11

Continued on next page

Table B.2 – Target List

OS	Applications
FreeBSD411	• 137/udp - Samba Samba 2.2.12 • 138/udp - Samba Samba 2.2.12 • 139/tcp - Samba Samba 2.2.12 • 25/tcp - Sendmail Consortium Sendmail 8.13.1 • 80/tcp - Apache Software Foundation Apache 1.3.33
FreeBSD41	• 25/tcp - Sendmail Consortium Sendmail 8.9.3
FreeBSD42	• 137/udp - Samba Samba 2.0.7 • 138/udp - Samba Samba 2.0.7 • 139/tcp - Samba Samba 2.0.7 • 25/tcp - Sendmail Consortium Sendmail 8.11.1
FreeBSD43	• 137/udp - Samba Samba 2.0.8 • 138/udp - Samba Samba 2.0.8 • 139/tcp - Samba Samba 2.0.8 • 25/tcp - Sendmail Consortium Sendmail 8.11.3 • 80/tcp - Apache Software Foundation Apache 1.3.9

Continued on next page

Table B.2 – Target List

OS	Applications
FreeBSD44	• 137/udp - Samba Samba 2.0.10 • 138/udp - Samba Samba 2.0.10 • 139/tcp - Samba Samba 2.0.10 • 25/tcp - Sendmail Consortium Sendmail 8.11.6 • 80/tcp - Apache Software Foundation Apache 1.3.20
FreeBSD45	• 137/udp - Samba Samba 2.2.2 • 138/udp - Samba Samba 2.2.2 • 139/tcp - Samba Samba 2.2.2 • 25/tcp - Sendmail Consortium Sendmail 8.11.6 • 80/tcp - Apache Software Foundation Apache 1.3.22
FreeBSD462	• 137/udp - Samba Samba 2.2.4 (Build 1) • 138/udp - Samba Samba 2.2.4 (Build 1) • 139/tcp - Samba Samba 2.2.4 (Build 1) • 25/tcp - Sendmail Consortium Sendmail 8.12.3
FreeBSD46	• 137/udp - Samba Samba 2.2.4 (Build 1) • 138/udp - Samba Samba 2.2.4 (Build 1) • 139/tcp - Samba Samba 2.2.4 (Build 1) • 25/tcp - Sendmail Consortium Sendmail 8.12.3 • 80/tcp - Apache Software Foundation Apache 1.3.24

Continued on next page

Table B.2 – Target List

OS	Applications
FreeBSD47	• 137/udp - Samba Samba 2.2.6 (Build pre2) • 138/udp - Samba Samba 2.2.6 (Build pre2) • 139/tcp - Samba Samba 2.2.6 (Build pre2) • 25/tcp - Sendmail Consortium Sendmail 8.12.6 • 80/tcp - Apache Software Foundation Apache 1.3.27
FreeBSD48	• 137/udp - Samba Samba 2.2.8 • 138/udp - Samba Samba 2.2.8 • 139/tcp - Samba Samba 2.2.8 • 25/tcp - Sendmail Consortium Sendmail 8.12.8 (Build p1) • 80/tcp - Apache Software Foundation Apache 1.3.27
FreeBSD49	• 137/udp - Samba Samba 2.2.8 a • 138/udp - Samba Samba 2.2.8 a • 139/tcp - Samba Samba 2.2.8 a • 25/tcp - Sendmail Consortium Sendmail 8.12.9 (Build p2) • 80/tcp - Apache Software Foundation Apache 1.3.28

Continued on next page

Table B.2 – Target List

OS	Applications
FreeBSD50	• 137/udp - Samba Samba 2.2.7 a • 138/udp - Samba Samba 2.2.7 a • 139/tcp - Samba Samba 2.2.7 a • 25/tcp - Sendmail Consortium Sendmail 8.12.6 • 80/tcp - Apache Software Foundation Apache 1.3.27
FreeBSD51	• 137/udp - Samba Samba 2.2.8 a • 138/udp - Samba Samba 2.2.8 a • 139/tcp - Samba Samba 2.2.8 a • 25/tcp - Sendmail Consortium Sendmail 8.12.9 • 80/tcp - Apache Software Foundation Apache 1.3.27
FreeBSD52	• 137/udp - Samba Samba 2.2.8 a • 138/udp - Samba Samba 2.2.8 a • 139/tcp - Samba Samba 2.2.8 a • 25/tcp - Sendmail Consortium Sendmail 8.12.10 • 80/tcp - Apache Software Foundation Apache 1.3.28

Continued on next page

Table B.2 – Target List

OS	Applications
FreeBSD53	• 137/udp - Samba Samba 2.2.12 • 138/udp - Samba Samba 2.2.12 • 139/tcp - Samba Samba 2.2.12 • 25/tcp - Sendmail Consortium Sendmail 8.13.1 • 80/tcp - Apache Software Foundation Apache 1.3.33
FreeBSD54	• 137/udp - Samba Samba 2.2.12 • 138/udp - Samba Samba 2.2.12 • 139/tcp - Samba Samba 2.2.12 • 25/tcp - Sendmail Consortium Sendmail 8.13.3 • 80/tcp - Apache Software Foundation Apache 1.3.33
LinuxFedora1Server	• 137/udp - Samba Samba 3.0 (Build 15) • 138/udp - Samba Samba 3.0 (Build 15) • 139/tcp - Samba Samba 3.0 (Build 15) • 21/tcp - Vsftpd Vsftpd 1.2.0 (Build 5) • 25/tcp - Sendmail Consortium Sendmail 8.12.10 (Build 1.1.1) • 445/tcp - Samba Samba 3.0 (Build 15) • 80/tcp - Apache Software Foundation Apache 2.0.47 (Build 10)

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxFedora2Server	• 137/udp - Samba Samba 3.0.3 (Build 5) • 138/udp - Samba Samba 3.0.3 (Build 5) • 139/tcp - Samba Samba 3.0.3 (Build 5) • 21/tcp - Vsftpd Vsftpd 1.2.1 (Build 5) • 25/tcp - Sendmail Consortium Sendmail 8.12.11 (Build 4.6) • 443/tcp - Apache Software Foundation Apache 2.0.49 (Build 4) • 445/tcp - Samba Samba 3.0.3 (Build 5) • 80/tcp - Apache Software Foundation Apache 2.0.49 (Build 4)
LinuxFedora3Server	• 137/udp - Samba Samba 3.0.8 (Build 0.pre1.3) • 138/udp - Samba Samba 3.0.8 (Build 0.pre1.3) • 139/tcp - Samba Samba 3.0.8 (Build 0.pre1.3) • 21/tcp - Vsftpd Vsftpd 2.0.1 (Build 5) • 25/tcp - Sendmail Consortium Sendmail 8.13.1 (Build 2) • 443/tcp - Apache Software Foundation Apache 2.0.52 (Build 3) • 445/tcp - Samba Samba 3.0.8 (Build 0.pre1.3) • 80/tcp - Apache Software Foundation Apache 2.0.52 (Build 3)

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxFedora4Server	• 137/udp - Samba Samba 3.0.14 a (Build 2) • 138/udp - Samba Samba 3.0.14 a (Build 2) • 139/tcp - Samba Samba 3.0.14 a (Build 2) • 21/tcp - Vsftpd Vsftpd 2.0.3 (Build 1) • 25/tcp - Sendmail Consortium Sendmail 8.13.4 (Build 2) • 443/tcp - Apache Software Foundation Apache 2.0.54 (Build 10) • 445/tcp - Samba Samba 3.0.14 a (Build 2) • 80/tcp - Apache Software Foundation Apache 2.0.54 (Build 10)
LinuxRH60	• 137/udp - Samba Samba 2.0.3 (Build 8) • 138/udp - Samba Samba 2.0.3 (Build 8) • 139/tcp - Samba Samba 2.0.3 (Build 8) • 21/tcp - Washington University wu-ftpd 2.4.2 VR17 (Build 3) • 25/tcp - Sendmail Consortium Sendmail 8.9.3 (Build 10) • 80/tcp - Apache Software Foundation Apache 1.3.6 (Build 7)
LinuxRH61	• 137/udp - Samba Samba 2.0.5 a (Build 12) • 138/udp - Samba Samba 2.0.5 a (Build 12) • 139/tcp - Samba Samba 2.0.5 a (Build 12) • 21/tcp - Washington University wu-ftpd 2.5.0 (Build 9) • 25/tcp - Sendmail Consortium Sendmail 8.9.3 (Build 15) • 80/tcp - Apache Software Foundation Apache 1.3.9 (Build 4)

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxRH62	• 137/udp - Samba Samba 2.0.6 (Build 9) • 138/udp - Samba Samba 2.0.6 (Build 9) • 139/tcp - Samba Samba 2.0.6 (Build 9) • 21/tcp - Washington University wu-ftpd 2.6.0 (Build 3) • 25/tcp - Sendmail Consortium Sendmail 8.9.3 (Build 20) • 80/tcp - Apache Software Foundation Apache 1.3.12 (Build 2)
LinuxRH70Server	• 137/udp - Samba Samba 2.0.7 (Build 21ssl) • 138/udp - Samba Samba 2.0.7 (Build 21ssl) • 139/tcp - Samba Samba 2.0.7 (Build 21ssl) • 21/tcp - Washington University wu-ftpd 2.6.1 (Build 6) • 25/tcp - Sendmail Consortium Sendmail 8.11 (Build 8) • 443/tcp - Apache Software Foundation Apache 1.3.12 (Build 25) • 445/tcp - Samba Samba 2.0.7 (Build 21ssl) • 80/tcp - Apache Software Foundation Apache 1.3.12 (Build 25)

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxRH71Server	• 137/udp - Samba Samba 2.0.7 (Build 38) • 138/udp - Samba Samba 2.0.7 (Build 38) • 139/tcp - Samba Samba 2.0.7 (Build 38) • 21/tcp - Washington University wu-ftpd 2.6.1 (Build 16) • 25/tcp - Sendmail Consortium Sendmail 8.11.2 (Build 14) • 443/tcp - Apache Software Foundation Apache 1.3.19 (Build 5) • 445/tcp - Samba Samba 2.0.7 (Build 38) • 80/tcp - Apache Software Foundation Apache 1.3.19 (Build 5)
LinuxRH72Server	• 137/udp - Samba Samba 2.2.1 a (Build 4) • 138/udp - Samba Samba 2.2.1 a (Build 4) • 139/tcp - Samba Samba 2.2.1 a (Build 4) • 21/tcp - Washington University wu-ftpd 2.6.1 (Build 18) • 25/tcp - Sendmail Consortium Sendmail 8.11.6 (Build 3) • 80/tcp - Apache Software Foundation Apache 1.3.20 (Build 16)
LinuxRH73Server	• 137/udp - Samba Samba 2.2.3 a (Build 6) • 138/udp - Samba Samba 2.2.3 a (Build 6) • 139/tcp - Samba Samba 2.2.3 a (Build 6) • 21/tcp - Vsftpd Vsftpd 1.0.1 (Build 5) • 25/tcp - Sendmail Consortium Sendmail 8.11.6 (Build 15) • 80/tcp - Apache Software Foundation Apache 1.3.23 (Build 11)

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxRH80	• 137/udp - Samba Samba 2.2.5 (Build 10) • 138/udp - Samba Samba 2.2.5 (Build 10) • 139/tcp - Samba Samba 2.2.5 (Build 10) • 21/tcp - Washington University wu-ftpd 2.6.2 (Build 8) • 25/tcp - Sendmail Consortium Sendmail 8.12.5 (Build 7) • 443/tcp - Apache Software Foundation Apache 2.0.40 (Build 8) • 445/tcp - Samba Samba 2.2.5 (Build 10) • 80/tcp - Apache Software Foundation Apache 2.0.40 (Build 8)
LinuxRH90	• 137/udp - Samba Samba 2.2.7 a (Build 7.9.0) • 138/udp - Samba Samba 2.2.7 a (Build 7.9.0) • 139/tcp - Samba Samba 2.2.7 a (Build 7.9.0) • 21/tcp - Vsftpd Vsftpd 1.1.3 (Build 8) • 25/tcp - Sendmail Consortium Sendmail 8.12.8 (Build 4) • 443/tcp - Apache Software Foundation Apache 2.0.40 (Build 21) • 445/tcp - Samba Samba 2.2.7 a (Build 7.9.0) • 80/tcp - Apache Software Foundation Apache 2.0.40 (Build 21)

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxSuSe10	• 137/udp - Samba Samba 3.0.20 (Build 4) • 138/udp - Samba Samba 3.0.20 (Build 4) • 139/tcp - Samba Samba 3.0.20 (Build 4) • 21/tcp - Vsftpd Vsftpd 2.0.3 (Build 6) • 25/tcp - Sendmail Consortium Sendmail 8.13.4 (Build 8) • 445/tcp - Samba Samba 3.0.20 (Build 4) • 80/tcp - Apache Software Foundation Apache 2.0.54
LinuxSuSe70	• 21/tcp - Washington University wu-ftpd 2.6.0 (Build 140) • 25/tcp - Sendmail Consortium Sendmail 8.10.2 • 80/tcp - Apache Software Foundation Apache 1.3.12
LinuxSuSe71	• 21/tcp - (Native) Linux 2.2.18-SUSE • 25/tcp - Sendmail Consortium Sendmail 8.11.2
LinuxSuSe72	• 21/tcp - (Native) Linux 2.4.4-4GB • 25/tcp - Sendmail Consortium Sendmail 8.11.3
LinuxSuSe73	• 21/tcp - (Native) Linux 2.4.10-4GB • 25/tcp - Sendmail Consortium Sendmail 8.11.6

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxSuSe80	• 137/udp - Samba Samba 2.2.3 a (Build 64) • 138/udp - Samba Samba 2.2.3 a (Build 64) • 139/tcp - Samba Samba 2.2.3 a (Build 64) • 21/tcp - Vsftpd Vsftpd 1.0.1 (Build 54) • 25/tcp - Sendmail Consortium Sendmail 8.12.2 (Build 88) • 80/tcp - Apache Software Foundation Apache 1.3.23 (Build 73)
LinuxSuSe81	• 137/udp - Samba Samba 2.2.5 • 138/udp - Samba Samba 2.2.5 • 139/tcp - Samba Samba 2.2.5 • 21/tcp - (Native) Linux 2.4.19-4GB • 25/tcp - Wietse Venema Postfix 1.1.11 (Build 88)
LinuxSuSe82	• 137/udp - Samba Samba 2.2.7 a • 138/udp - Samba Samba 2.2.7 a • 139/tcp - Samba Samba 2.2.7 a • 21/tcp - (Native) Linux SUSE82 2.4.20-4GB • 25/tcp - Wietse Venema Postfix 2.0.6 (Build 8)

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxSuSe90	• 137/udp - Samba Samba 2.2.8 a (Build 107) • 138/udp - Samba Samba 2.2.8 a (Build 107) • 139/tcp - Samba Samba 2.2.8 a (Build 107) • 21/tcp - (Native) Linux SUSE90 2.4.21-99-defaulf • 25/tcp - Sendmail Consortium Sendmail 8.12.10 (Build 7) • 80/tcp - Apache Software Foundation Apache 1.3.28 (Build 43)
LinuxSuSe91	• 139/tcp - Samba Samba 3.0.2 a (Build 51) • 21/tcp - PureFTPd PureFTPd 1.0.18 (Build 35) • 25/tcp - Sendmail Consortium Sendmail 8.12.10 (Build 158) • 445/tcp - Samba Samba 3.0.2 a (Build 51) • 80/tcp - Apache Software Foundation Apache 2.0.49
LinuxSuSe92	• 137/udp - Samba Samba 3.0.7 (Build 5) • 138/udp - Samba Samba 3.0.7 (Build 5) • 139/tcp - Samba Samba 3.0.7 (Build 5) • 21/tcp - Vsftpd Vsftpd 2.0.1 (Build 2) • 25/tcp - Sendmail Consortium Sendmail 8.13.1 (Build 5) • 445/tcp - Samba Samba 3.0.7 (Build 5) • 80/tcp - Apache Software Foundation Apache 2.0.50

Continued on next page

Table B.2 – Target List

OS	Applications
LinuxSuSe93	• 137/udp - Samba Samba 3.0.12 (Build 5) • 138/udp - Samba Samba 3.0.12 (Build 5) • 139/tcp - Samba Samba 3.0.12 (Build 5) • 21/tcp - Vsftpd Vsftpd 2.0.2 (Build 3) • 25/tcp - Sendmail Consortium Sendmail 8.13.3 (Build 5) • 445/tcp - Samba Samba 3.0.12 (Build 5) • 80/tcp - Apache Software Foundation Apache 2.0.54
NetBSD151	• 137/udp - Samba Samba 2.0.8 • 138/udp - Samba Samba 2.0.8 • 139/tcp - Samba Samba 2.0.8 • 25/tcp - Sendmail Consortium Sendmail 8.11.3 • 80/tcp - Apache Software Foundation Apache 1.3.19
NetBSD153	• N/A
NetBSD161	• N/A
NetBSD162	• N/A
NetBSD16	• N/A

Continued on next page

Table B.2 – Target List

OS	Applications
OpenBSD26	• 137/udp - Samba Samba 2.0.5 a • 138/udp - Samba Samba 2.0.5 a • 139/tcp - Samba Samba 2.0.5 a • 25/tcp - Sendmail Consortium Sendmail 8.9.3 • 80/tcp - Apache Software Foundation Apache 1.3.9
OpenBSD27	• 139/tcp - Samba Samba 2.0.6 • 25/tcp - Sendmail Consortium Sendmail 8.10.1 • 80/tcp - Apache Software Foundation Apache 1.3.12
OpenBSD28	• 137/udp - Samba Samba 2.0.7 • 138/udp - Samba Samba 2.0.7 • 139/tcp - Samba Samba 2.0.7 • 25/tcp - Sendmail Consortium Sendmail 8.10.1 • 80/tcp - Apache Software Foundation Apache 1.3.12
OpenBSD29	• 137/udp - Samba Samba 2.0.8 • 138/udp - Samba Samba 2.0.8 • 139/tcp - Samba Samba 2.0.8 • 25/tcp - Sendmail Consortium Sendmail 8.11.3 • 80/tcp - Apache Software Foundation Apache 1.3.19

Continued on next page

Table B.2 – Target List

OS	Applications
OpenBSD30	• 189/tcp - Samba Samba 2.2.1 a • 25/tcp - Sendmail Consortium Sendmail 8.12.1 • 80/tcp - Apache Software Foundation Apache 1.3.19
OpenBSD31	• 137/udp - Samba Samba 2.2.3 a • 138/udp - Samba Samba 2.2.3 a • 139/tcp - Samba Samba 2.2.3 a • 25/tcp - Sendmail Consortium Sendmail 8.12.2 • 80/tcp - Apache Software Foundation Apache 1.3.24
OpenBSD32	• 137/udp - Samba Samba 2.2.5 • 138/udp - Samba Samba 2.2.5 • 139/tcp - Samba Samba 2.2.5 • 25/tcp - Sendmail Consortium Sendmail 8.12.6 • 80/tcp - Apache Software Foundation Apache 1.3.26
OpenBSD33	• 137/udp - Samba Samba 2.2.8 • 138/udp - Samba Samba 2.2.8 • 139/tcp - Samba Samba 2.2.8 • 25/tcp - Sendmail Consortium Sendmail 8.12.9 • 80/tcp - Apache Software Foundation Apache 1.3.27

Continued on next page

Table B.2 – Target List

OS	Applications
OpenBSD34	• 137/udp - Samba Samba 2.2.8 a • 138/udp - Samba Samba 2.2.8 a • 139/tcp - Samba Samba 2.2.8 a • 25/tcp - Sendmail Consortium Sendmail 8.12.9 • 80/tcp - Apache Software Foundation Apache 1.3.28
OpenBSD35	• 137/udp - Samba Samba 2.2.9 • 138/udp - Samba Samba 2.2.9 • 139/tcp - Samba Samba 2.2.9 • 25/tcp - Sendmail Consortium Sendmail 8.12.11 • 80/tcp - Apache Software Foundation Apache 1.3.29
Win2000SP1SSL	• N/A
Win2000SP1	• 21/tcp - Microsoft IIS FTP Server 5.0 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60 • 80/tcp - Microsoft IIS Web Server 5.0
Win2000SP2NetDDE	• N/A
Win2000SP2SSL	• N/A

Continued on next page

Table B.2 – Target List

OS	Applications
Win2000SP2	• 21/tcp - Microsoft IIS FTP Server 5.0 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60 • 80/tcp - Microsoft IIS Web Server 5.0
Win2000SP3NetDDE	• N/A
Win2000SP3SSL	• N/A
Win2000SP3	• 21/tcp - Microsoft IIS FTP Server 5.0 • 25/tcp - Microsoft IIS SMTP Service 5.0 • 80/tcp - Microsoft IIS Web Server 5.0
Win2000SP4NetDDE	• N/A
Win2000SP4SSL	• N/A
Win2000SP4	• 21/tcp - Microsoft IIS FTP Server 5.0 • 80/tcp - Microsoft IIS Web Server 5.0

Continued on next page

Table B.2 – Target List

OS	Applications
Win2000ServerSP1	• 21/tcp - Microsoft IIS FTP Server 5.0 • 25/tcp - Microsoft IIS SMTP Service 5.0 • 80/tcp - Microsoft IIS Web Server 5.0
Win2000ServerSP2	• 21/tcp - Microsoft IIS FTP Server 5.0 • 25/tcp - Microsoft IIS SMTP Service 5.0 (Build 2195.2966) • 80/tcp - Microsoft IIS Web Server 5.0
Win2000ServerSP3	• 21/tcp - Microsoft IIS FTP Server 5.0 • 25/tcp - Microsoft IIS SMTP Service 5.0 • 80/tcp - Microsoft IIS Web Server 5.0
Win2000ServerSQLSP1	• 1433/tcp - Microsoft SQL Server 2000 SP1 • 1434/udp - Microsoft SQL Server 2000 SP1
Win2000ServerSQLSP2	• 1433/tcp - Microsoft SQL Server 2000 SP2 • 1434/udp - Microsoft SQL Server 2000 SP2
Win2000ServerSQLSP3	• 1433/tcp - Microsoft SQL Server 2000 SP3 • 1434/udp - Microsoft SQL Server 2000 SP3

Continued on next page

Table B.2 – Target List

OS	Applications
Win2000ServerSQLSP3a	• 1433/tcp - Microsoft SQL Server 2000 SP3a • 1434/udp - Microsoft SQL Server 2000 SP3a
Win2000ServerSQLSP4	• 1433/tcp - Microsoft SQL Server 2000 SP4 • 1434/udp - Microsoft SQL Server 2000 SP4
Win2000ServerSQL	• 1433/tcp - Microsoft SQL Server 2000 • 1434/udp - Microsoft SQL Server 2000
Win2000Server	• 21/tcp - Microsoft IIS FTP Server 5.0 • 25/tcp - Microsoft IIS SMTP Service 5.0 (Build 2172.1) • 80/tcp - Microsoft IIS Web Server 5.0
Win2000	• 21/tcp - Microsoft IIS FTP Server 5.0 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60 • 80/tcp - Microsoft IIS Web Server 5.0
Win95	• 21/tcp - TYPSoft TYPSoft FTP Server 1.08

Continued on next page

Table B.2 – Target List

OS	Applications
Win98SE	• 21/tcp - Fastream NetFile FTP/Web Server 5.9.0.562 • 23/tcp - Midasoft 123 Terminal Server 1.2-1628 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60
Win98	• 21/tcp - TYPSoft TYPSoft FTP Server 1.08 • 23/tcp - Midasoft 123 Terminal Server 1.2-1628 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60
WinMe	• 21/tcp - Fastream NetFile FTP/Web Server 5.9.0.562 • 23/tcp - Midasoft 123 Terminal Server 1.2-1628 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60
WinNT43Server	• 21/tcp - Microsoft IIS FTP Server 3.0 • 80/tcp - Microsoft IIS Web Server 3.0
WinServer2003SP1	• 21/tcp - Microsoft IIS FTP Server 6.0 • 25/tcp - Microsoft IIS SMTP Service 6.0 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60 • 80/tcp - Microsoft IIS Web Server 6.0

Continued on next page

Table B.2 – Target List

OS	Applications
WinServer2003	• 21/tcp - Microsoft IIS FTP Server 6.0 • 25/tcp - Microsoft IIS SMTP Service 6.0 (Build 3790.0) • 80/tcp - Microsoft IIS Web Server 6.0
WinXPHomeSP1a	• 69/udp - SolarWinds TFTP Server Standard Edition 5.3.23
WinXPHomeSP2	• 69/udp - SolarWinds TFTP Server Standard Edition 5.3.23
WinXPHome	• 69/udp - SolarWinds TFTP Server Standard Edition 5.3.23
WinXPProSP1aNetDDE	• N/A
WinXPProSP1a	• 21/tcp - Microsoft IIS FTP Server 6.0 • 25/tcp - Microsoft IIS SMTP Service 6.0 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60 • 80/tcp - Microsoft IIS Web Server 6.0
WinXPProSP2	• 21/tcp - Microsoft IIS FTP Server 6.0 • 25/tcp - Microsoft IIS SMTP Service 6.0 • 69/udp - SolarWinds TFTP Server Standard Edition 5.0.60 • 80/tcp - Microsoft IIS Web Server 6.0

Continued on next page

Table B.2 – Target List

OS	**Applications**
WinXPPro	• 21/tcp - Microsoft IIS FTP Server 5.1 • 80/tcp - Microsoft IIS Web Server 5.1