A HYBRID APPROACH TO OPERATING SYSTEM DISCOVERY BASED ON DIAGNOSIS THEORY

by

François Gagnon

fgagnon@sce.carleton.ca

Ph.D. Thesis Proposal

School of Computer Science

CARLETON UNIVERSITY

Ottawa, Ontario

January, 2009

Table of Contents

List of Tables

List of Figures

Abstract

Motivated by the need to filter non-critical intrusion detection alarms (alarms that a not indicative of a plausible threat), we study how contextual information could be used to achieve this task. We start by establishing that using information about the configuration of the target (operating system and application version) is extremely useful in identifying non-critical alarms. Then, we demonstrate that current operating system discovery (OSD) tools are not adequate for gathering the required information. After identifying the reasons for their inadequacy, we propose a new OSD approach and a partial implementation of our approach. Our approach is more flexible, less intrusive, and promises to be more accurate than the existing ones. Moreover, unlike existing OSD tools which are completely ad hoc, we adopt a formal approach and develop our tool according to the principles of diagnosis problem solving. This formalization allows us to characterize the complexity of OSD and to adapt algorithms designed for diagnosis. To address the needs of OSD, we extend the theory of diagnosis with a query-based approach. This leads to a study of several algorithms to solve different queries. Finally, we provide a virtual environment allowing to gather OS fingerprints in a systematic and automated way.

Today is where your book begins, the rest is still unwritten.

-Natasha Bedingfield (Unwritten)

Chapter 1

Introduction

A person should be able to do a small bit of everything, specialization is for insects.
-Unknown Source

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. 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. Examples of situations where such information is useful are:

- Assuring compatibility of the network with new infrastructures (software or hardware).
- Helping technical support teams by providing automated troubleshooting.
- Assisting network security officers by filtering out *non-critical* alarms (i.e., alarms not indicative of a plausible threat).

Throughout this thesis, our principal motivation will be security. However, the approach will be general enough so it can be used in other contexts.

1.1 Motivation

Signature-based intrusion detection systems (IDSes) detect malicious packets by matching them against a predefined set of rules. Even though they usually produce a high number of non-critical alarms, they are still widely used as a primary defense system for networks. Non-critical alarms are problematic in two ways:

- A security officer could spent a considerable amount of time discarding non-critical alarms instead of investigating real threats.

- Automatically preventing attacks from compromising the target based on IDS alarms requires that the IDS does not produce too many false alarms as this could disrupt normal traffic.

Given an IDS alarm, we want to decide whether it is relevant (i.e., whether the corresponding attack will result in compromising the target). To make such a decision, one should consider the context surrounding the attack. For instance, information about the configuration of the target and the network topology can be seen as part of that context. The operating system running on the target is the primary piece of information regarding its configuration. For instance, an exploit meant to compromise a Microsoft IIS web server launched against a Linux computer is harmless and would fail, but it could still trigger an alarm if the packet matches some detection rules.

As a matter of fact, an experiment (see Chapter 2) showed that by considering only the operating systems of the targets, assuming we know the eact OS of each target, we could eliminate nearly half of the non-critical alarms while not eliminating any critical ones. Unfortunately, using existing OSD tools to gather this knowledge provides very poor results (less than 13% of non-critical alarms are eliminated in the best case).

Moreover, it is unrealistic to assume that a static knowledge base containing the configuration of each computer could be maintained by a network administrator as networks are becoming more and more dynamic and heterogeneous. Thus, a better operating system discovery tool is needed for IDS context gathering.

1.2 Problem Statement

In this thesis we propose to design a new tool for operating system discovery. We have three main objectives regarding the development of our tool:

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

- We want to study the computational complexity of OSD and take advantages of effective and well-tested algorithms. Both can be achieved by adopting an existing theoretical 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.

1.3 Thesis Structure

The thesis is structured as follows:

Chapter 2 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 2 confirms the usefulness of operating system information for IDS context and shows that existing OSD tools are not adequate for gathering the information needed.

Chapter 3 introduces operating system discovery and presents the two classical approaches: *active* (where probes can be sent to the target to stimulated a reaction) and *passive* (where no packet can be injected). It also presents a state of the art of OSD tools. Of particular importance is the discussion about the advantages and limitations of current OSD tools/approaches, see Section 3.6. This discussion provides insight as to why current OSD tools are not adequate for IDS context gathering.

Chapter 4 states the problem addressed through this thesis and details our principal objectives.

Chapter 5 presents the ideas of our new knowledge-oriented *hybrid* approach to OSD, its partial implementation (currently only the passive module is operational) and experimental results. The early experimental results are extremely promising as the tool is already more accurate than every other passive tool and is also better than well known active tools such as Nmap [78]. The current implementation relies on *Answer Set Programming* as a knowledge representation language and greatly suffers from long computation time. We are currently working on improving the implementation. In order to work in the right direction, it is important to understand the inherent complexity of the different tasks related to OS discovery.

With the objective of understanding the complexity of OSD, Chapter 6 introduces

the theory of logical diagnosis and explains how operating system discovery can be seen as a diagnosis task. Based on the theory of diagnosis, we propose a fast algorithm for the passive module of our tool (much more efficient than the initial algorithm of Chapter 5). Moreover, Chapter 7 explains why the current diagnosis framework is not flexible enough for OSD and argues for a query-based approach to diagnosis. This leads to a detailed study of test selection strategies for solving the different queries, and ultimately to implementing the active module of our tool. These two chapters provide the theoretical basis of our proposed tool.

Finally, our third objective, providing a systematic and automated way of collecting OS fingerprints, is addressed in Chapter 8. This chapter presents VNEC, a virtual network experiment controller. This tool is used to automatically gather traffic traces from which operating system fingerprints can easily be extracted. Those fingerprints can then be automatically transformed into rules, thus providing a fully automated way of updating our OSD tool when new OSes are released.

Chapter 9 provides a summary of the work proposed to complete this thesis. It highlights the remaining deliverables and sketches a timeline for the estimated completion of each of them.

1.4 Contributions

The main contribution of this thesis is a *hybrid* approach to operating system discovery combining both the *passive* and *active* techniques in a knowledge-oriented way by using a diagnosis framework. We attack the OSD problem from the 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 also, we want to have a good understanding of the underlying complexity and reuse existing algorithms. In this case, logical diagnosis will provide an adequate theory to represent OSD..

it to be supported by a strong knowledge representation and theoretical model in order to reuse generic results, and

Moreover, we have three other major contributions:

- Since our primary motivation is security, we provide a study of the impact of contextual information for the reduction of non-critical alarms in intrusion

detection systems. In particular, we measure the usefulness of knowledge about the target operating system to support IDS and establish that existing OSD tools are not good enough to gather that information.

- We extend the theory of diagnosis in order to obtain the flexibility required by our OSD tool. 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.

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

1.5 Collaboration

Part of the work in this thesis has been done in collaboration with the Network Security research group at the Canada Communication Research Center (CRC). More precisely, the IDS context evaluation experiments of Chapter 2 were executed in their lab and the virtual environment discussed in Chapter 8 is an evolution of the one available at CRC. I would like to thank Frédéric Massicotte, from CRC, for making this fruitful collaboration possible.

Chapter 2

Motivation - Using Operating System Discovery to Gather Intrusion Detection Context

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

Abstract

This chapter analyzes the relevance of target configuration (i.e. operating system and applications) as contextual information to reduce the number of noncritical alarms produced by an IDS. It also studies the performance of current tools for operating system and application discovery for the task of IDS context gathering. The experiment conducted shows that knowing the configuration of the target can help filtering out over 70% of noncritical alarms without dropping true positives. However, the same experiment demonstrates that current tools for OS discovery are not accurate enough to filter out the majority of noncritical alarms.

2.1 Introduction

It has been pointed out in [43, 46] that one of the main drawbacks of signature-based intrusion detection systems (IDSes) is the large amount of *noncritical alarms*[1] they produce. Those alarms pose two problems. First, a security officer might spend all of his time discarding noncritical alarms instead of investigating real threats. Second, in order to automatically prevent attacks based on IDS alarms, one must be confident that this will not disrupt the normal activities of the network; that is, legitimate traffic should not be dropped due to false alarms.

In order to reduce the number of noncritical alarms, researchers have proposed to consider the context of the attack to establish whether it has any chance to succeed. Some commercial products are currently developed to integrate contextual information into IDSes. A good example is Sourcefire's RNA [65]; using both network and

[1] alarms not related to a successful attack, see Definition 2.2

target configuration information. However, very little has been done to assess the relevance of a given piece of information as the context for noncritical alarm reduction and how such information should be gathered to better serve intrusion detection.

In this chapter, we focus on assessing the effectiveness of using target configuration information to reduce noncritical alarms. By target configuration we mean the target operating system (OS) version and the version of the applications (App) offering the services available on the target open ports. We demonstrate, using a large scale experiment, how relevant this information is and we compare the two pieces of information (`OS` and `App`) to see which one is more useful and whether one of them is sufficient. Moreover, we consider the task of gathering this information and test how well currently available tools for operating system and application discovery perform when used for IDS context gathering.

The rest of this chapter is structured as follows: Section 2.2 starts by defining the alarm classes considered and the concept of non-critical alarm. Then Section 2.3 provides a survey and classification of different approaches to IDS context. Section 2.4 details the experiment setup used to determine the effectiveness of target configuration. Section 2.5 provides an upper bound on the number of non-critical alarms that can be removed by target configuration (when we know the exact configuration of the targets) while Section 2.6 provides the results when current tools are used to gather the target configuration information. Finally, Section 2.7 will wrap up the chapter with a discussion, while Section 2.8 will highlight the chapter's contributions.

2.2 Alarm Classes

IDSes are purely reactive in the sense that they act only in the presence of network events (usually IDSes react at the packet level, but they could also react at the communication session level). Thus whenever an IDS raises an alarm, this alarm is related to the event which triggered the IDS reaction.

Most of the time, an IDS alarm is simply classified as a true positive (when the underlying event is a successful attack, i.e., it succeeds in compromising the target) or a false positive (otherwise). However, this classification is very coarse and, for instance, does not distinguish between a failed malicious attempt and normal traffic

erroneously flagged by an IDS; both being classified as false positives. They are both false positive because they do not result in successful attack. But one (the failed malicious attempt) is of interest, because it indicate the presence of an attacker; while the other (normal traffic erroneously flagged) is just annoying.

Here, we consider the finer-grained alarm type structure based on the following characterization of the event space, see Figure 2.1:

- Is the event normal or malicious? Here, we simply split the event in two, malicious events and normal events. Note, however, that the distinction between the two is not always clear. In fact, there is no formal definition of a malicious event. Here, we informally consider an event as malicious if the intent of its perpetrator was dishonest.

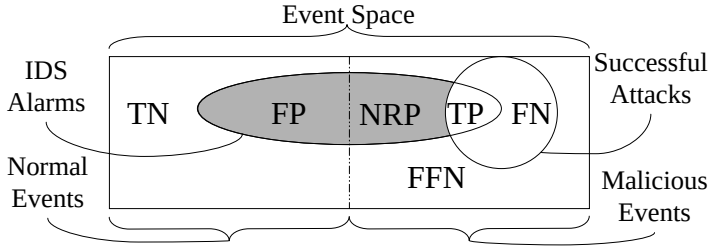
- If the event is malicious, did the attack fail or succeed? Even if the event is malicious, it is not necessarily the case that the attack succeeded. In fact, there are several reasons why a malicious event would fail to compromise a target:

  - Some packets don't make it to the target, they are blocked by some network device (e.g., a firewall or a router if the TTL is too low).
  - Some packet will not be accepted by the target (e.g., not part of an established TCP session).
  - Some packet will be interpreted differently by the target and the IDS (e.g., not all OSes will handle overlapping packet fragments in the same way, see [64, 69]).
  - The target may not be vulnerable (e.g., it has been patched for a known vulnerability).
  - etc.

  Note that since we consider IDS, and not IPS (intrusion prevention system), the packets are never blocked and always make it the to target. Thus it is possible, in theory, to know whether the attack succeeded or not.

- Is the event flagged or not by the IDS? Do we have an alarm associated to the event?

Figure 2.1: Alarm Types



Based on the above characteristics (normal vs malicious, successful vs failed, and detected vs undetected), we consider the six alarm classes presented in Definition 2.1.

Definition 2.1 (Alarm Types)
We consider the following six alarm types, see Figure 2.1:

TN: *A True Negative is a normal (non-malicious) event undetected by the IDS. Negative means the IDS did not consider the event as malicious, while true means the IDS was correct.*

FN: *A False Negative is a malicious event that succeeded in compromising the target and was undetected by the IDS.*

FFN: *A Fortuitous False Negative is a malicious event that failed to compromise the target and was undetected by the IDS.*

TP: *A True Positive is a successful malicious event detected by the IDS.*

FP: *A False Positive is a normal event for which the IDS raised an alarm.*

NRP: *Following [40], a Non-Relevant Positive is a malicious event which failed to compromise the target but was detected by the IDS.* ◯

Based on the above alarm types, we can formally define the concept of noncritical alarms.

Definition 2.2 (Noncritical Alarm)
An alarm is noncritical when it is not related to a successful attack attempt. That is, noncritical alarms = FP ∪ NRP (shaded area of Figure 2.1). ◯

2.3 Survey of Contextual Approaches

In the past few years, researchers have proposed several different contextual solutions to reduce the number of noncritical alarms generated by signature-based IDSes. The four principal approaches being:

- Considering networking features, Section 2.3.1.1
- Using target configuration, Section 2.3.1.2
- Relying on vulnerability assessments, Section 2.3.1.3
- Monitoring attack side effects, Section 2.3.1.4

This section first introduces the approaches, classifies them, and finally provides a brief survey of existing work related to the evaluation of contextual information.

2.3.1 Introducing Contextual Approaches

Contextual information should help provide the following information:

- Will the packet reach the target?
- Will the packet be accepted by the target?
- How will the target interpret the packet?
- How will the target react after consuming the packet?

The goal of gathering this information is to help an IDS answer the question "Will the target be compromised after consuming the packet?".

The focus here is entirely on network-based solutions where context is gathered through the network. Other solutions, like host-based or even manually created static knowledge bases, are not discussed. Although potentially more accurate, the later are much more difficult to deploy and maintain in large and heterogeneous environments. Moreover, the level of target configuration information available on public vulnerability repositories (such as SecurityFocus) is limited (e.g., vulnerable OSes are not defined in terms of patch version, but only in terms of version and service pack).

2.3.1.1 Network-Related Context

Network-related context contains networking information (e.g. network topology, communication protocol specifications, firewall awareness, etc) that can help decide whether an attack attempt has any chances of succeeding. For instance, if a malicious packet detected by an IDS cannot reach its target (e.g., the TTL value is too small, it will be stopped by firewall rules, etc[2]) , then it is obvious that the attempt will fail. Historically, the importance of considering context in intrusion detection was first noted in [54] where the authors proposed to consider malicious TCP packets only when part of a valid TCP session in order to prevent *squealing*. Squealing consists of sending many individual packets that will match a signature and thus trigger an alarm (a false alarm), but that are harmless because they are not part of a valid TCP session. This process can help to hide a real attack within thousands of false alarms. The use of valid TCP session information has since been embedded in most signature-based IDSes to prevent this kind of squealing attacks.

Taking into account network-related context addresses two questions: will the packet reach the target and will the packet be accepted by the target?

2.3.1.2 Target Configuration Information

Although target configuration could be seen as a special case of vulnerability assessment (see Section 2.3.1.3), they are presented independently because they differ in the level of information they deal with (`OS` and `App` instead of vulnerability to a particular flaw).

[69] discusses how knowing the target operating system can help resolve the ambiguity concerning the way a target will interpret the packets (for instance, how the target deals with overlapping TCP segments). More generally, we can use target configuration information to infer whether the target is vulnerable to a given attack by looking at vulnerability databases such as SecurityFocus [63]. Figure 2.2 shows an entry from SecurityFocus for BID 1163. This entry represents a vulnerability in NetBIOS for some version of Windows failing to handler specific conditions. It is stated that only Windows 98 and Windows 95 are vulnerable, while no product is

[2] see [72]

specifically listed as being not vulnerable.

Figure 2.2: SecurityFocus Entry for BID 1163

Microsoft Windows 9x NetBIOS NULL Name Vulnerability	
Bugtraq ID:	1163
Class:	Failure to Handle Exceptional Conditions
CVE:	N/A
Remote:	yes
Local:	yes
Published:	May 02 2000 12:00AM
Updated:	May 02 2000 12:00AM
Credit:	Posted to Bugtraq on May 2, 2000 by Evan Brewer
Vulnerable:	Microsoft Windows 98
	Microsoft Windows 95
Not Vulnerable:	

The work presented in [22] uses `OS` information as context directly in Snort rules. However, the authors only consider the target operating system (not the application), and they completely discard noncritical alarms instead of assigning them a lower priority. They developed a Snort plugin to detect the operating systems and modified the rules to raise an alarm only when the target has a vulnerable OS. Bro [55] also has a mechanism to specify that an alarm should only be raised when the target configuration meets certain criteria, see [67]. For instance, it is common with Bro to have a condition stating that the target web server must be running Microsoft IIS for an alarm to be raised[3]. A major problem with these two approaches is that when we do not have any information about the target configuration, the IDS rule will not match and no alarm will be created. As mentioned in [45] this implies that Bro will miss some successful attacks. The approach presented in [22] presumably suffers from the same problem.

Considering target configuration information mainly addresses two questions: how

[3]This is the case when Bro is using Snort rules converted with the s2b rule translator.

will the target interpret the packet and will the target be compromised after consuming the packet?

2.3.1.3 Vulnerability Assessment

Vulnerability assessment determines whether the target is vulnerable to the security flaw exploited by the current attack. [40] proposes to use Nessus [70] as a vulnerability scanner to determine whether a system has a given vulnerability. Two different strategies are mentioned:

- Scanning the network every week (or every month) for vulnerabilities. However, the vulnerability information may not be accurate anymore when an alarm occurs, and the system could mistakenly classify a critical alarm as being unimportant.

- Actively scanning the network when an attack occurs. However, this might significantly increase the traffic load on the network. Moreover, we can either choose to hold the suspicious packet back until the scan is done or the let it go right away (and execute the scan after a possible attack). Temporarily blocking the packet could disrupt the network activity, since a vulnerability scan can take some time. On the other hand, if the scan is done after the attack, the system may have already been compromised (or may be down) and the scanner could be fooled by the attacker (in thinking that the system is not vulnerable).

While vulnerability assessment is possibly very accurate, it requires the repeated use of a vulnerability scanner which creates traffic on the network. This traffic can interact in an inconvenient way with network monitoring tools (IDSes usually detect vulnerability scans as they often precede an attack). Furthermore, a vulnerability scan might unintentionally compromise the target computer (see the "What Makes Systems Crash?" section of [62]).

Vulnerability assessment directly addresses the following question: will the target be compromised after consuming the packet?

2.3.1.4 Attack Side Effects

Considering the attack side effects means inspecting the target behavior for evidences that the attack was successful. This mainly allows to answer the following question: how does the target react after consuming the packet?

[67] (using Bro) and [81] (using Snort) propose to specify the target reaction directly in the IDS rules. For a given attack, an alarm should be raised only when a predefined target reaction is observed. For instance, we may know that a specific web attack fails when the server replies with an error message (e.g. HTTP 400 Bad Request). [67] and [81] encode such a reaction with a rule saying that an alarm should be raised only when the server responds with a message that is not an error message[4]. While the intended semantics of such a rule is to capture the fact that the server respond with an error message whenever the attempt failed, the signature actually needs a reply to assess that the server did respond with a non-error message. Thus, no alarm is raised when the target does not reply (maybe because the attack attempt crashed the target). As mentioned in [45], this implies Bro will miss some successful attacks. The approach presented in [81] presumably suffers from the same problem.

The work presented in [40] proposes a different use of the attack side effect. The authors suggest to remotely access an agent on the target after an attack attempt to gather evidences of the attack success/failure (log files, host-based IDSes, etc). Although this approach provides detailed information, it would be hard to deploy and maintain in large heterogeneous networks.

There seems to be a major drawback to the side effect approach: in order to get side effects, one must let the attack execute on the target. Thus, when a reliable alarm is generated, the attack has already been fully executed. Moreover, when the attack attempt crashes the target, it might not be possible to witness any side effects.

[4]It is not possible to do otherwise since the signature language of Snort (and most likely the one of Bro) does not allow to specify that a specific event does not occur (e.g. no response of a given type from the target).

2.3.1.5 Alarm Correlation

[28, 49] present another mechanism, called alarm correlation, to reduce the amount of alarms generated by IDSes. Basically, alarm correlation engines take as input events from different sources and identify how these events are mutually related. The main goal is to reduce the number of alarms, by merging all the alarms related to a common intrusion attempt into a single alarm. This usually leads to more meaningful alarms. Even though this technique is not explicitly an enhancement through context, the framework can be used to correlate the alarms of an IDS with the contextual information (that would be provided by a third party tool seen as a source distinct from the IDS). We also know from [45] that Bro and Snort have complementary quality for detection and verification. Thus, it makes sense to expect that events from both IDSes can be correlated to remove some false alarms.

2.3.2 Classifying Contextual Approaches

In this section we propose properties that can be used to compare and select the best contextual approach. Comparison is not only about effectiveness (which the rest of this chapter is about) but also about the "cost" of gathering the information required. We believe the cost of an approach depends on the following properties:

- Is the information required static or dynamic? If the information is static, we need to gather it only once (maybe manually), while if it is dynamic, we need to gather it several times or at least keep it up-to-date. Thus dynamic information will require more effort.

- Can we obtain the info before the attack occurs, or must we do it after? If we don't have to wait for the attack to take place, we have the possibility to prevent the attack, by blocking the event. If it must be done after the attack, then we cannot prevent the attack.

- Is it possible to gather the information passively, i.e., simply by monitoring the activities on the network? If it is not possible, then the cost of gathering the information is higher as we must take actions on the network. If we have to (or choose to) rely on active techniques, then:

- Can it be done in a non-intrusive way? It is better if the actions blend into the normal activities of the network, otherwise the information gathering process could itself trigger alarms.
- Can it be done in a safe way? It is preferable that the actions be harmless to the target system, after all, we don't want to crash the target ourselves just to find out that the "real" attack attempt failed.

Below we discuss the properties of the four contextual approaches presented earlier, see also Table 2.1 for a summary.

Table 2.1: Comparing Contextual Approaches

	Network Information	**Target Configuration**	**Vulnerability Assessment**	**Attack Side Effect**
Static/ Dynamic	Static	Nearly Static	Nearly Static	N/A
a priori/ a posteriori	a priori	a priori	a priori	a posteriori
Passive/ Active	Manual & passive	Passive & active	Active	Passive & active
Intrusive	N/A	Variable	Yes	No
Safe	N/A	Yes	Not entirely	Yes

2.3.2.1 Network-Related Context

Network-related context is mainly about static information. Protocol specifications (e.g., TCP three-way handshake) are entirely static, while a network topology (e.g., to evaluate TTL) is mostly static.

There is no need to let the attack take place to obtain the required information as the information is totaly independent of the attack outcome.

Due to the nature of the information, part of it is gathered manually (protocol specification); the rest (network topology) can be gathered passively, see [41].

The network-related context approach cost effective. However, it has a very limited accuracy (as it handles only very obvious cases) and is usually not enough by itself.

2.3.2.2 Target Configuration Information

Target configuration context relies on near static information. However, changes in a computer configuration (e.g., new OS, new IP, new applications) do occur sometimes and thus the information is not entirely static.

Once again, there is no need to let the attack take place to obtain the required information as the information is totaly independent of the attack outcome. The information can be gathered with a combination of active and passive techniques.

Target configuration has a pretty low cost, especially if we emphasize passive monitoring.

2.3.2.3 Vulnerability Assessment

Vulnerability assessment also relies on somewhat static information. However, vulnerability changes will occur more frequently than target configuration changes: there are several security updates available for a given version of Windows (e.g., for Windows 2000 Sp4), these produce a change in the vulnerability, but not in the OS itself.

Again, there is no need to let the attack take place to obtain the information.

Existing tools to gather vulnerability information are active, intrusive and can even compromise the target, see [62].

Vulnerability assessment is probably the most effective approach, as it directly gather the pertinent information. However, it is also the most costly.

2.3.2.4 Attack Side Effects

The attack side effect inspection approach relies on a posteriori information (meaning the attack must occur before we can fetch relevant information). Thus it cannot be classified into dynamic vs static information.

Part of the information can be gathered passively (e.g., protocol error messages, see [67] and [81]), while the rest has to be gathered actively (e.g., consult the target log file, see [40]).

This method is fairly new, so its effectiveness is hard to gauge, but one drawback of this approach is that the attack must occur for the context to be available.

2.3.3 Evaluating Contextual Approaches

So far, not much effort has been made towards evaluating the effectiveness of a given piece of information as context and most evaluation experiments are ad hoc. New, systematic and large scale, evaluation experiments are important to measure the real impact these approaches have on alarm filtering, to understand how different approaches can be combined together, and to provide direction for further research. This section presents a brief overview of the different ad hoc evaluation experiments published.

[22] reports an evaluation of the effectiveness of using operating system context based on three large real traffic traces. The problem with large real traffic traces is the difficulty (or impossibility) to confirm that the results are corrects. The results in [22] simply indicate a 5% to 10% reduction of the number of alarms; it is not clear whether they verified that the alarms suppressed were indeed noncritical alarms (precision) and whether the alarms that were not suppressed were all important (recall).

[40] presents two evaluations for their vulnerability assessment approach. The first uses traffic traces generated by launching exploits in an experimental network, while the second one uses traffic recorded near some honeypots. In both cases, the classification verification was done manually. It is mentioned that manual inspection was possible since most attacks targeted non-existent services. Since the inspection was manual, it is nearly impossible to reproduce the experiment and harder still to perform the same experiment on a new dataset. Moreover, since most attacks targeted non-existent services, it is obvious that the machines were not vulnerable and most attacks would fail, thus we do not have an idea about how many real attacks would be discarded or tagged as harmless. Thus, it is not clear whether this approach would be reliable in a more realistic environment.

[81] performs an evaluation of the attack side effect approach using traffic collected near three honeypots. The classification results were manually verified for three months of traffic traces. Since only three different targets were used, it is not possible to see the extend to which target reaction is dependent on the target configuration ([45] mentions that target reaction is dependant on the target configuration).

The evaluation we propose here is different since the process is fully automatic.

We also propose a more complete evaluation, considering different classification algorithms and comparing the best case scenarios (when we have complete knowledge of the context) with real-world scenarios (using tools to gather the context). The evaluation is done using a wide variety of attack scenarios and targets. Our evaluation will allow us to confirm the potential of target configuration information for reducing non-critical alarms. Moreover, it will give us a measure of the effectiveness of current OS discovery tools for gathering this information. Based on that, we will derive the main objectives of this thesis.

2.4 Experiment Setup

The main goal of this chapter is to evaluate the relevance of `OS` and `App` information to eliminate noncritical alarms. To achieve this, the experiment is divided in three parts: dataset generation, information management, and alarm classification (Sections 2.4.1, 2.4.2, and 2.4.3 respectively).

2.4.1 Dataset Generation

To measure the effectiveness of using target configuration information for IDS context we need a dataset of attack scenarios. Instead of building a whole dataset from scratch, we used the one presented in [45] which is freely available. Every traffic trace contained in this dataset is the result of launching a Vulnerability Exploitation Program (VEP) against a target[5]. The dataset contains 92 easily accessible VEPs implementing 47 different well-known vulnerabilities, see Table A.1 in Appendix A. Each vulnerability corresponds to a Bugtraq ID (BID) referring to the vulnerability database of SecurityFocus [63]. The 47 vulnerabilities included in the dataset are diverse in the sense that they are distributed over 15 different ports, including the common ports: 21, 23, 25, and 80. Moreover, these VEPs are launched against a pool of 95 targets with different popular operating systems and/or applications, see Table A.2 in Appendix A for OS details. We believe this dataset is representative of an average hacker since the VEPs are available on popular sites such as SecurityFocus and the vulnerable products are popular OSes and/or applications.

[5]Sometimes a VEP is used several times, with different option settings, against a target.

More details about the environment used to generate the data set can be found in Section 8.3.1 of Chapter 8 and in [45].

2.4.1.1 Using the Dataset

The interesting aspect of this dataset is the documentation of the traffic traces: for each trace we know, among other things, the BID of the VEP used, the port targeted, the operating system and applications installed on the target, and more importantly the outcome of the attack attempt (success or failure). Thus it is possible to retrieve the vulnerable products (OSes and applications) for the current attack (using the BID and SecurityFocus) and compare with the actual target's OS and App to decide whether the attack has any chance of succeeding. Then, we can automatically verify whether we correctly classified the alarm by considering the actual outcome of the attempt. Another important point is that we have access to the actual targets and the environment in which the dataset was created. This is provided by the Communication Research Center (CRC) of Canada.

Each traffic trace is given to Snort and the alarms are recorded. To simplify the experiment and the results analysis, the alarms not directly related to the VEP are eliminated. Thus, for each traffic trace, only the alarm that is related to the attack, if any, is kept. Therefore, Snort will never produce false positives. It will only generate true positives and non-relevant positives (see Figure 2.1). So the experiment will measure the ability of `OS` and `App` information to eliminate non-relevant positives. The next assumption follows from such a design choice.

Assumption 2.1
Snort only raises alarms that are directly related to the attack attempt.

Even though this assumption clearly does not hold in reality, the results obtained here do reflect the ability of `OS` and `App` information to eliminate non-relevant positives. Moreover, it seems reasonable to assume that target configuration would also be able to remove false positives, although the amount it would remove is unknown.

2.4.2 Information Management

The information used for the experiment can be summarized into three database tables:

OS information: for each target in the dataset, we have an entry associating the IP address to the operating system of that computer (e.g. 10.1.1.3 is running Windows XP Sp1).

App information: for each target used in the dataset and for each open port on that target, we have an entry associating the IP address and the open port to the application running on that port (e.g. 10.1.1.3 is running Microsoft IIS FTP Server 5.0 on port 21/tcp).

SecurityFocus vulnerability information: for each BID used in the dataset and for every product, we have an entry indicating if the product is vulnerable, not vulnerable, or unspecified[6] (e.g. Linux RedHat 7.0 is the only vulnerable product for BID 3335 and since nothing is listed as non-vulnerable all other products are unspecified).

This is the information that will be used to decide whether an alarm is a true positive and should be kept or it is noncritical and should be discarded. In fact, we do not discard noncritical alarms, we classify them as "likely to fail", see Section 2.4.3.

The `OS` and `App` information was gathered manually when the targets were created and the applications installed. Thus, we know the exact version of these products. This gives the following experiment assumption:

Assumption 2.2

The exact target configuration is available.

It is not clear whether this assumption is realistic. However, it is consistent with the goal of the experiment: in order to assess the relevance of the `OS` and `App` information for IDS context, we want to know whether this context can be useful in the best conditions. Section 2.6 will relax this assumption by using currently available

[6] as mentioned by SecurityFocus

tools to dynamically gather the context information instead of assuming total a priori knowledge.

We chose to use SecurityFocus as the source for vulnerability information based on our past fruitful experiences and the fact that there is a strong referencing link between Snort's alarms and the SecurityFocus BIDs. According to our experience, one limitation of SecurityFocus is the lack of distinction between type of products (OS, App, and others). For instance, the only vulnerable product for BID 11820 is "Microsoft w3who.dll" which is neither an OS nor an App (as it is not a service offered on a port). However, we are not aware of any vulnerability sources providing a better product classification.

2.4.3 Alarm Classification

The goal of using context information is to prevent non-critical alarms from overloading the security officer. However, simply erasing non-critical alarms would not be a good idea. A better approach is to classify alarms into

Likely to Succeed (LS): There is some evidence that the attack mentioned by this alarm will succeed.

Likely to Fail (LF): There is some evidence that the attack mentioned by this alarm will fail (false or non-relevant positives).

Attempt (A): The alarm cannot be classified due to contradictory or lack of evidence.

The idea is that alarms should not be deleted, but classified (assigned priority levels). This allows a security officer to select which classes of alarms he wants to see in real-time and still give him the opportunity to check the lower priority alarms for further forensic. For instance, a security officer could select to receive only LS and A alarms in real-time. Once in a while, he may choose to take a look at the logged LF to discover that someone is systematically trying to exploit a vulnerability in the `sadmin` service of Sun Solaris, but against Windows machines (those attempts would be LF). From now on, he could start closely monitoring the activities of this attacker.

Unfortunately, we rapidly found out that it is not reliable to classify an alarm as LS based on the fact that the target is running a vulnerable OS and/or App. This is not surprising as there are many (contextual) conditions that are essential for an attack to succeed. All of them must be met for the attack to succeed, but the attack fails as soon as one of them is not respected. So, we decided to focus only on classifying alarms as LF (when there are evidences that the attack will fail) and A (otherwise).

2.4.4 Classification Algorithms

We implemented four different decision procedures to test whether an alarm should be classified as LF. This section details those algorithms and explains their role in the experiment.

Each algorithm takes as input an alarm "a" ($a.x$ denotes the value of attribute x for the alarm a) and outputs a class for the alarm (A or LF). *pOS(M)* denotes the set of operating systems possibly running on machine M and *pApp(M,P)* denotes the set of applications possibly running on port P of machine M. As discussed previously in Section 2.4.2, we assume that we always know the actual product, thus those sets will in fact be singletons (this assumption is lifted in Section 2.6). Furthermore, *V(B)* (resp. *NV(B)*) denotes the set of products listed as vulnerable (resp. non-vulnerable) by SecurityFocus for the BID B.

2.4.4.1 ContextOS algorithm

The first algorithm (ContextOS) considers only the `OS` information and works as follows, see also Figure 2.3:

(1) by default, assign A;

(2) if the target OS is listed as non-vulnerable for this exploit, then LF;

(3) if the target OS is not listed as vulnerable and

- **(3.1)** the products listed as vulnerable for the exploit are all OSes, then LF.

Condition (3.1) is important because a system can be vulnerable even though its OS is not listed as vulnerable (for instance if the vulnerable products for the BID are applications).

Figure 2.3: ContextOS Algorithm

```
ContextOS(a)
class ← A                                   (1)
IF pOS(a.target) ⊆ NV(a.bid)                (2)
   class ← LF
ELSE IF pOS(a.target) ∩ V(a.bid) = ∅        (3)
   IF V(a.bid) ⊆ OS                         (3.1)
      class ← LF
RETURN class
```

In the case where we don't know one of the products listed as vulnerable on SecurityFocus (that is, we don't know whether it is an application or an operating system), we pessimistically assume that it is both an application and an operating system:

Assumption 2.3
Unknown vulnerable products are always pessimistically considered to be of both types (OS and App).

This assumption will disable the case (3.1) whenever there is an unknown vulnerable product for the current BID. This will give a pessimistic approximation, but it is important to preserve the soundness of the classification procedure. When implementing the use of context in an IDS, one could make sure that he knows the type of every SecurityFocus products. However, this would be tedious and we believe the pessimistic assumption will closely approximate a real world implementation. Moreover, we implemented a different version of the ContextOS algorithm where the above assumption was replaced by its converse optimistic assumption (losing the soundness). The results (not presented here) were not sufficiently better than those with the pessimistic assumption to be of any concern.

2.4.4.2 ContextApp Algorithm

The second algorithm (ContextApp) considers only the `App` information and is very similar to the previous one, see Figure 2.4. We still have the pessimistic Assumption 2.3. With this algorithm and the previous one, it is possible to compare `OS` and `App` information.

Figure 2.4: ContextApp Algorithm

ContextApp(a)

```
class ← A                                          (1)
IF pApp(a.target) ⊆ NV(a.bid)                      (2)
   class ← LF
ELSE IF pApp(a.target) ∩ V(a.bid) = ∅              (3)
   IF V(a.bid) ⊆ App                               (3.1)
      class ← LF
RETURN class
```

2.4.4.3 ContextOSApp Algorithm

The third algorithm (ContextOSApp) considers both the `OS` and `App` information simultaneously, see Figure 2.5. The goal here is to see what is the gain of having both pieces of information. There is no need for the pessimistic assumption 2.3 anymore. Unknown products can safely be discarded since for each target we know the exact OS and App installed.

2.4.4.4 ContextOSDeductions Algorithm

The last algorithm (ContextOSDeductions) is an extension to the one presented in Section 2.4.4.1 and also considers only the `OS` information, see Figure 2.6 where cannotRunOn(θ) provides a set of applications that cannot run on any of the operating systems in θ. It uses a static deduction table to determine which popular applications cannot run on the target given the OS. For instance, if the target is

Figure 2.5: ContextOSApp Algorithm

ContextOSApp(a)

class ← A (1)
IF pOS(a.target) ⊆ NV(a.bid) OR pAPP(a.target) ⊆ NV(a.bid) (2)
  class ← LF
ELSE IF pOS(a.target) ∩ V(a.bid) = ∅ AND pApp(a.target) ∩ V(a.bid) = ∅ (3)
  class ← LF
RETURN class

Linux Red Hat 7.0, then it is not possible that Microsoft IIS FTP Server 5.0 runs on port 21/tcp.

The goal of this algorithm is to see whether we can enhance the performance of `OS` information to the point where application information becomes practically irrelevant. This is based on the belief that gathering the `OS` information is both easier and more reliable than gathering and maintaining the `App` information for all the open ports.

Figure 2.6: ContextOSDeductions Algorithm

ContextOSDeductions(a)

class ← A (1)
IF pOS(a.target) ⊆ NV(a.bid) (2)
  class ← LF
ELSE IF pOS(a.target) ∩ V(a.bid) = ∅ (3)
  IF V(a.bid) ⊆ OS (3.1)
    class ← LF
  ELSE IF V(a.bid) ⊆ OS ∪ App (3.2)
    IF (V(a.bid) ∩ App) ⊆ cannotRunOn(pOS(a.target)) (3.2.1)
      class ← LF
RETURN class

2.5 Results - Complete Knowledge

This section presents the results of the experiment for the case where we know the exact configuration of each target. Among the 47 vulnerabilities contained in the dataset, 7 are totally undetected by Snort (by BID: 11372, 14513, 2010, 3064, 5311, 5411, and 6005). Thus we are left with 40 BIDs for which it is possible to filter noncritical alarms.

Before discussing the results, Section 2.5.1 first describes the performance measures used to analyze the results. Then, Section 2.5.2 discusses the very few classification mistakes made by the algorithms (classifying TP as LF). Experimental results are presented in Section 2.5.3.

2.5.1 Performance measures

The goal of the classification algorithms presented in Section 2.4.4 is to find the noncritical alarms among all available alarms. This can be viewed as an information retrieval task. For this reason, we use the classic measures of information retrieval to assess the performance of the algorithms. We mainly use precision and recall as defined below. The perfect algorithm would have a recall of 100% (it is able to classify every noncritical alarm as likely to fail) while having a precision of 100% (it does not classify any true positive as likely to fail).

Precision: the number of alarms correctly classified as likely to fail divided by the number of alarms classified as likely to fail.

Recall: the number of alarms correctly classified as likely to fail divided by the number of noncritical alarms.

2.5.2 Precision

There are 5 vulnerabilities for which at least one classification algorithm has a precision of less then 100% (it made a mistake classifying some true positives as likely to fail), see precision details for the algorithms in Table 2.2 (a summary can be found in Table 2.4, second column). At first this is surprising since the algorithms of Section

2.4.4 are sound. However, after further investigation, it turned out that those mistakes can be explained by the fact that some apparently vulnerable products are not listed as such on SecurityFocus. They are (by BID):

1163: Windows 98 SE is not listed as vulnerable, but the attack did work against this OS.

3335: Linux Red Hat 6.0, 6.1, 6.2, 7.1, 7.2, 7.3 and Linux SuSe 8.0, 10.0 are not listed as vulnerable, however the attack did work against these OSes.

3581: The default FTP servers installed on Suse 7.2 and 7.3 should be vulnerable. This requires further investigation.

514: Windows 98 SE is not listed as vulnerable, however the attack did work against this OS.

9633: Windows 2000 SP4 Pro is not listed as vulnerable, but the attack was successful.

Table 2.2: Precision for the Four Context Algorithms

Algo / **BID**	**Context OS**	**Context App**	**Context OSDeductions**	**Context OSApp**
10078	0/ 0	149/ 149	209/ 209	285/ 285
10108	30/ 30	0/ 0	30/ 30	30/ 30
10115	22/ 22	0/ 0	22/ 22	22/ 22
10116	80/ 80	0/ 0	80/ 80	80/ 80
1163	51/ 52	0/ 0	51/ 52	51/ 52
11763	0/ 0	0/ 0	0/ 0	0/ 0
11820	0/ 0	0/ 0	40/ 40	0/ 0
1331	15/ 15	0/ 0	15/ 15	15/ 15
1578	0/ 0	3/ 3	0/ 0	3/ 3
1806	0/ 0	182/ 182	140/ 140	182/ 182

Continued on next page

Table 2.2 – Precision Details - continued from previous page

Algo / BID	**Context OS**	**Context App**	**Context OSDeductions**	**Context OSApp**
2124	12/ 12	0/ 0	39/ 39	168/ 168
2417	70/ 70	0/ 0	70/ 70	70/ 70
2503	0/ 0	29/ 29	0/ 0	29/ 29
2674	0/ 0	156/ 156	120/ 120	156/ 156
2708	0/ 0	224/ 224	180/ 180	224/ 224
2906	27/ 27	0/ 0	27/ 27	27/ 27
307	0/ 0	35/ 35	20/ 20	35/ 35
3335	26/ 34	0/ 0	26/ 34	26/ 34
3581	0/ 0	33/ 33	21/ 21	68/ 70
4006	0/ 0	0/ 0	0/ 0	1/ 1
4482	0/ 0	0/ 0	0/ 0	136/ 136
4485	0/ 0	0/ 0	0/ 0	103/ 103
5033	0/ 0	0/ 0	0/ 0	53/ 53
514	133/ 136	0/ 0	133/ 136	133/ 136
529	0/ 0	27/ 27	20/ 20	27/ 27
5556	0/ 0	0/ 0	0/ 0	12/ 12
7106	0/ 0	67/ 67	117/ 117	184/ 184
7116	0/ 0	0/ 0	0/ 0	81/ 81
7230	0/ 0	0/ 0	0/ 0	2/ 2
7294	0/ 0	32/ 32	0/ 0	283/ 283
754	53/ 53	0/ 0	53/ 53	53/ 53
8035	124/ 124	0/ 0	124/ 124	124/ 124
8205	0/ 0	0/ 0	0/ 0	81/ 81
8315	0/ 0	0/ 0	113/ 113	281/ 281
8459	6/ 6	0/ 0	6/ 6	6/ 6
8615	35/ 35	0/ 0	35/ 35	35/ 35
8826	4/ 4	0/ 0	4/ 4	4/ 4

Continued on next page

Table 2.2 – Precision Details - continued from previous page

Algo / BID	**Context OS**	**Context App**	**Context OSDeductions**	**Context OSApp**
9633	11/ 12	0/ 0	11/ 12	11/ 12
9635	0/ 0	0/ 0	0/ 0	52/ 52
9751	0/ 0	111/ 111	156/ 156	213/ 213

2.5.3 Recall

Now that it is clear that the algorithms would not misclassify a true positive as likely to fail (they have 100% precision) as long as the vulnerability database is reliable, let's see how well they perform in classifying non-relevant positives as likely to fail (recall). Table 2.3 presents the recall for each algorithm on each BID. We can see that ContextOSApp is the best algorithm; it is also the one requiring the largest amount of knowledge. Obviously, ContextOSDeductions is never worse than ContextOS. But it is surprising[7] to see how much better it is than ContextOS: for twelve BIDs, ContextOSDeductions is better than ContextOS. It is harder to compare ContextOSDeductions with ContextApp since one is not always better than the other. For some BIDs (20) ContextOSDeductions is better than ContextApp, but for some others (10) ContextApp is better. It is harder still to compare ContextOS with ContextApp based only on Table 2.3.

Table 2.3: Recall for the Four Context Algorithms

Algo / BID	**Context OS**	**Context App**	**Context OSDeductions**	**Context OSApp**
10078	0/ 285	149/ 285	209/ 285	285/ 285
10108	30/ 59	0/ 59	30/ 59	30/ 59

Continued on next page

[7]Let's recall that the deductions are entirely based on static knowledge.

Table 2.3 – Recall Details - continued from previous page

Algo / BID	**Context OS**	**Context App**	**Context OSDeductions**	**Context OSApp**
10115	22/ 53	0/ 53	22/ 53	22/ 53
10116	80/ 180	0/ 180	80/ 180	80/ 180
1163	51/ 52	0/ 52	51/ 52	51/ 52
11763	0/ 2	0/ 2	0/ 2	0/ 2
11820	0/ 70	0/ 70	40/ 70	0/ 70
1331	15/ 15	0/ 15	15/ 15	15/ 15
1578	0/ 10	3/ 10	0/ 10	3/ 10
1806	0/ 221	182/ 221	140/ 221	182/ 221
2124	12/ 172	0/ 172	39/ 172	168/ 172
2417	70/ 70	0/ 70	70/ 70	70/ 70
2503	0/ 33	29/ 33	0/ 33	29/ 33
2674	0/ 194	156/ 194	120/ 194	156/ 194
2708	0/ 261	224/ 261	180/ 261	224/ 261
2906	27/ 32	0/ 32	27/ 32	27/ 32
307	0/ 35	35/ 35	20/ 35	35/ 35
3335	26/ 26	0/ 26	26/ 26	26/ 26
3581	0/ 69	33/ 69	21/ 69	68/ 69
4006	0/ 4	0/ 4	0/ 4	1/ 4
4482	0/ 148	0/ 148	0/ 148	136/ 148
4485	0/ 139	0/ 139	0/ 139	103/ 139
5033	0/ 69	0/ 69	0/ 69	53/ 69
514	133/ 133	0/ 133	133/ 133	133/ 133
529	0/ 28	27/ 28	20/ 28	27/ 28
5556	0/ 12	0/ 12	0/ 12	12/ 12
7106	0/ 262	67/ 262	117/ 262	184/ 262
7116	0/ 120	0/ 120	0/ 120	81/ 120
7230	0/ 2	0/ 2	0/ 2	2/ 2

Continued on next page

Table 2.4: Precision/Recall Summary for the Four Algorithms

Measure / Algo	**Overall Precision**	**Overall Recall**
ContextOS	699/712 (98.17%)	699/4575 (15.28%)
ContextApp	1048/1048 (100.00%)	1048/4575 (22.91%)
ContextOSDeductions	1862/1875 (99.31%)	1862/4575 (40.70%)
ContextOSApp	3346/3361 (99.55%)	3346/4575 (73.14%)

Table 2.3 – Recall Details - continued from previous page

Algo / BID	**Context OS**	**Context App**	**Context OSDeductions**	**Context OSApp**
7294	0/ 386	32/ 386	0/ 386	283/ 386
754	53/ 53	0/ 53	53/ 53	53/ 53
8035	124/ 136	0/ 136	124/ 136	124/ 136
8205	0/ 370	0/ 370	0/ 370	81/ 370
8315	0/ 466	0/ 466	113/ 466	281/ 466
8459	6/ 23	0/ 23	6/ 23	6/ 23
8615	35/ 35	0/ 35	35/ 35	35/ 35
8826	4/ 32	0/ 32	4/ 32	4/ 32
9633	11/ 18	0/ 18	11/ 18	11/ 18
9635	0/ 87	0/ 87	0/ 87	52/ 87
9751	0/ 213	111/ 213	156/ 213	213/ 213

To get a general view of the algorithms' efficiency, Table 2.4 (third column) presents the overall recall for each algorithm on the whole dataset. This gives a general idea of how well the algorithms performed on the dataset. ContextApp (23%) performs better than ContextOS (15%) and ContextOSDeductions (41%) performs much better than ContextApp (23%). ContextOSApp (73%) has the best results.

2.5.4 Discussion

The results presented here are clear: by filtering more than 70% of the non-critical alarms, target configuration has a lot of potential. Moreover, `OS` and `App` are both useful and they are complementary.

2.6 Results - Tools for Context Gathering

Section 2.5 presented the potential of target configuration as contextual information to reduce non-relevant alarms; that is, assuming the exact target configuration information is available. In this section, we relax assumption 2.2 and experiment using current tools to gather the contextual information. Section 2.6.1 presents how current tools for operating system discovery (OSD) perform; while Section 2.6.2 discusses the performance of current tools for application discovery (AppD).

2.6.1 Operating System Discovery Tools

We tested several passive OSD tools (Siphon 0.666beta [68], SinFP 2.00-8 [3], ettercap NG-0.7.3 [50], the 4 modes of p0f 2.0.8 [80]) as well as two active tools (Xprobe 2.0.3 [2] and Nmap 4.20 [78]). We used the ContextOSDeductions algorithm of Section 2.4.4.4, but now considering a set of possible OSes (as provided by the OSD tools). For the passive tools, the set of possible OSes is the set of OSes guessed by the tool when using the corresponding attack trace as input[8]. For the active tools, we ran the tool once for each target and used the results for every attack against that target.

2.6.1.1 Precision

Once again, the tools achieve almost 100% precision, see Table 2.5. Some errors are still explained by the fact that some apparently vulnerable products are not listed as such on SecurityFocus. However, another source of error here is potential incorrect guesses by OSD tools which could result in a true positive being classified as likely to fail. In this case, a target is considered non-vulnerable, since we have the wrong OS,

[8]So we limit the traffic seen by the passive tools to the traffic generated by the exploit.

Table 2.5: Precision/Recall Summary for OSD Tools

Measure / OSD Tool	Overall Precision	Overall Recall
p0f (Syn)	1/1 (100.00%)	1/4575 (2.19%)
p0f (StrayAck)	29/31 (93.55%)	29/4575 (0.63%)
SinFP	75/75 (100.00%)	75/4575 (1.64%)
Siphon	109/113 (96.46%)	109/4575 (2.38%)
p0f (RstAck)	156/156 (100.00%)	156/4575 (3.41%)
Nmap	232/236 (98.31%)	232/4575 (5.07%)
ettercap	387/392 (98.72%)	387/4575 (8.46%)
p0f (SynAck)	399/414 (96.38%)	399/4575 (8.72%)
Xprobe	583/589 (98.98%)	583/4575 (12.74%)
exact OS (see Table 2.4)	1862/1875 (99.31%)	1862/4575 (40.70%)

while in fact it is. Precision details for OSD tools are given in Tables A.3 to A.11 in Appendix A.

2.6.1.2 Recall

Detailed recall results for the tested OSD tools are presented in Tables A.3 to A.11 in Appendix A. It also includes, for comparison purpose, the recall score for the case where we know the exact OS. There is not a huge gap between the best passive tool and the best active tool (9% vs 13%). However, it is clear that OSD tools still have a long way to go to allow a significant reduction of non-relevant positives, the best tool only achieves one third of the target configuration information potential (13% vs 40%).

2.6.2 Application Discovery Tools

We tested one active tool (Nmap 4.20) and one passive tool (ettercap NG-0.7.3) for application discovery. We used the ContextApp algorithm; considering the set of possible applications for the target as given by the tools.

2.6.2.1 Precision

The ContextApp algorithm did not make any mistakes on the BIDs considered here. Thus, any mistake reported by an application discovery tools is the result of an incorrect application guess. Surprisingly enough, both tools have a precision over 99.5%; see Table 2.6 (details can be found in Appendix A). However, for the 12 BIDs considered here, five do not have any successful attacks (for those five, the precision is automatically 100%). We believe that on a larger scale, application discovery tools would have a lower precision.

2.6.2.2 Recall

Both application discovery tools perform very well in that they fully exploit the potential of considering application information in reducing non-critical alarms. The recall results, see Table 2.6 (details of recall are presented in Table A.12 and A.13 in Appendix A), show that Nmap performs better than the ideal case (i.e, when we actually know the exact App). This might seem strange, but can be explained as follows. Assume the target App is vulnerable to a given attack, but the attack failed due to other reasons (e.g., the target did not accept the malicious packet, or the exploit required the target application to run as root and this was not the case). The classification algorithm with full knowledge would not classify the attack as likely to fail (since the application is vulnerable), therefore it has 0/1 recall. Now, suppose Nmap makes a mistake while detecting the underlying application and it concludes that the target is not vulnerable ending up with a 1/1 recall. In that example, Nmap was fortunate, as its mistake was rewarded. Usually, this kind of mistake should result in a decrease of precision. However, in every BID where Nmap guesses incorrectly, we do not have a single successful attempt. This is a limitation of the dataset we are using.

2.6.3 Discussion

Once again, the results are quite clear. Current OS discovery tools are not adequate for IDS context gathering as they don't even achieve half their potential in non-critical

Table 2.6: Precision/Recall Summary for AppD Tools

Measure / AppD Tool	Overall Precision	Overall Recall
ettercap	863/865 (99.77%)	863/4575 (18.86%)
exact App (see Table 2.4)	1048/1048 (100.00%)	1048/4575 (22.91%)
Nmap	1238/1240 (99.84%)	1238/4575 (27.06%)

alarm reduction. On the other hand, application discovery tools seem good enough, although it is not yet clear that their precision would not be an issue.

2.7 Conclusion

The experiment conducted here clearly shows that the use of target configuration as contextual information is an effective approach for IDS alarm reduction. In the ideal case and by combining both `OS` and `App` information we can filter out over 70% of non-critical alarms. However, current tools for operating system discovery are definitely not adequate for IDS context gathering, not even achieving half of the potential. On the other hand, application discovery tools are good enough.

The work presented in this chapter has a few limitations:

- As stated by Assumption 2.1, we currently only measure the ability to reduce non-critical positives. We can only conjecture that the approach would also be useful to eliminate false positives.

- As mentioned in Section 2.6.2.1, the precision results of the application discovery tools (and maybe of the os discovery tools as well) might not be accurate. This is due to the limited number of successful attempts in the dataset.

- Another limitation comes from the absence of a comparison between target configuration and the other contextual approaches. It is possible that vulnerability assessment would provide better information than target configuration. However, based on the properties of each approach, we believe that target configuration should be used as much as possible while other techniques, such as

vulnerability assessment and attack side effect should only be used as a complement to target configuration.

2.8 Contributions

The main contribution of this chapter is an experiment measuring the relevance of target configuration as contextual information for an IDS as well as the effectiveness of current tools to gather that context. Other contributions include:

- The definition of important properties related to contextual approaches.
- A classification of the different contextual approaches based on those properties.
- A survey of the few existing experiments used to evaluate IDS context.
- The elaboration of a new, large scale, experiment dedicated to the evaluation of contextual approaches.

Chapter 3

State of the Art - Operating System Discovery

Everyone who got where he is had to begin where he was.
-Robert Louis Stevenson

Abstract

This chapter builds on two results of the previous chapter: target configuration is a relevant piece of information for IDS context and current operating system discovery tools are not good enough for that task. From there, we formally introduce the problem of operating system discovery. We also present a detailed study of the current approaches (active and passive) as well as some existing tools. Finally, we discuss what are the limitations of these approaches and why they are not well-suited for context gathering.

3.1 Introduction

It is increasingly difficult for an administrator (or a program) to monitor a computer network for possible problems. For instance, the once simple task of keeping track of the operating systems running on the networked computers is now tedious and time consuming. The importance of knowing what are the operating systems (OS) running on the network is substantial: assuring compatibility with software or hardware, providing technical support, and preventing security breaches are few examples where such knowledge is essential. Chapter 2 provided a good example where this information could be useful, i.e., reducing non-critical IDS alarms.

Operating system discovery (OSD) is the task of figuring out which operating system (OS) is running on networked computers. One way of doing it would be to manually keep a database containing that information up to date. However, this is a tedious task as networks are growing bigger and user have more and more control over their computers (to install and update OSes). A more scalable way is to analyze the communication behavior of a computer, more specifically the content of the communications, and deduce the operating system running on that computer. In the rest of

this thesis, we focus on the second approach, since it can be entirely automatic. The basic idea is that in many situations, there is no unique standardized way for an OS to behave in response to a given packet, and each OS constructor must implement the behavior of their choice (or must simply respect some vague guidelines). For instance, the time-to-live (TTL) field of an IP packet must be set to a value sufficiently large to avoid dropping too many packets, see [8]. Although the current recommended value for TTL is 64, see [61], it is not mandatory to use that specific number; thus different vendors may use different values (e.g. Windows ues a TTL of 32 or 128 while Linux uses a TTL of 64 or 255).

Historically, two type of OS discovery tools have been developed: passive and active. An *active* tool sends specific stimuli to the target and analyzes the way it responds. A *passive* tool, on the other hand, will simply use the packets that are available on the network to find out the OS of each computer. These two approaches are presented in details in Section 3.4 and 3.3 respectively.

The rest of this chapter is structured as follows. First, Section 3.2 describes the type of tests that can be used to discover the operating system. Next, Section 3.3 presents the passive approach as well as some passive tools. Section 3.4 introduces the active approaches and some active tools. Section 3.6 discusses the limitations of these two approaches. Section 3.7 briefly mentions countermeasures to prevent OS discovery and argues why their existence do not affect the task of gathering OS information for IDS context. Finally, Section 3.5 presents the result of an experiment to compare the accuracy of some existing OSD tools.

3.2 OS Discovery Tests

Both the active and passive approaches rely on *tests* to perform discovery. Following [1], 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 executed as single-packet tests (considering only the response); this will result in some loss of information, see Example 3.1.

Sample: A sample test is a test requiring several packets sent from the target in order to extract its behavior.

Example 3.1 (stimulus-response tests regarded as single-packet tests) Consider a simplified version of the TCP RstAck stimulus-response test (see Test-9 in Definition B.9) where we only inspect the "don't fragment" (DF) bit (which can be either set of not) of both stimulus and 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 don't, than we conclude it is Windows 2000 sp4. If we were to consider this test as 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. ♢

Table 3.1 provides a brief description of all the OSD tests considered in this thesis (inspired from [1]). For each test, we provide its type (as introduced above), whether it relies entirely on well-formed traffic or if it uses some malformed packets[1] and whether it can be performed passively, actively or both (see discussion below). A detailed description of each test is given in Appendix B.

[1]A packet is malformed if it does not respect the protocol standard, e.g., A TCP packet with no flags set. Note that a malformed packet is not necessarily malicious, but it could be detected by an IDS nevertheless.

Table 3.1: OS Discovery Tests

Property / **Test**	**Type**	**Well-formed/ Malformed**	**Active/ Passive**
Test-1 TCP Syn	Single-Packet	Well-formed	Passive
Test-2 ARP Request	Single-Packet	Well-formed	Both
Test-3 TCP ISN	Sample	Well-formed	Passive
Test-4 IP ID	Sample	Well-formed	Both
Test-5 TCP TS	Sample	Well-formed	Both
Test-6 ARP Retransmit	Sample	Well-formed	Both
Test-7 ICMP ID Seq	Sample	Well-formed	Passive
Test-8 SynAck	Stimulus-Response	Well-formed	Both
Test-9 RstAck	Stimulus-Response	Well-formed	Both
Test-10 ICMP Unreach	Stimulus-Response	Well-formed	Both
Test-11 ICMP Echo	Stimulus-Response	Well-formed	Both
Test-12 ICMP Info	Stimulus-Response	Well-formed/ Malformed	Both
Test-13 ICMP TS	Stimulus-Response	Well-formed	Both

Continued on next page

Table 3.1 – OS Discovery Tests

Property / Test	**Type**	**Well-formed/ Malformed**	**Active/ Passive**
Test-14 ICMP Mask	Stimulus-Response	Well-formed	Both
Test-20 SynEcn	Stimulus-Response	Well-formed	Active
Test-21 no flags	Stimulus-Response	Malformed	Active
Test-22 SynFinUrgPsh	Stimulus-Response	Malformed	Active
Test-23 Ack open	Stimulus-Response	Well-formed	Active
Test-24 Ack closed	Stimulus-Response	Well-formed	Active
Test-25 FinUrgPsh	Stimulus-Response	Well-formed	Active
Test-26 Echo Request	Single Packet	Well-formed	Passive

A test is entirely passive if it relies on spontaneous events, i.e., events that cannot be triggered on-demand. As a consequence, it is not possible to trigger such an event actively. For instance, Test-1 relies on the first packet of a TCP handshake and it is not possible to trigger this event using a stimulus event. On the other hand, a test is entirely active if it relies on a reactive event (the response to a stimulus) that will not occur on a normal network (usually the stimulus is a malformed packet). As a consequence, it is not possible to observe this event passively. For instance, Test-21 analyzes the response of the target to a TCP packet with no flags. According to the TCP protocol specification[2], every TCP packet should have at least one flag.

[2]According to RFC 793, there is no semantics associated with a TCP packet with no flags.

Finally, tests that are both active and passive relies 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 the response to a TCP SYN request. This request happens often in a network, but it can also be injected on-demand

From now on, we say a test is malformed (resp. well-formed) if it partially (resp. totally) relies on malformed (resp. well-formed) traffic.

3.3 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, it is not possible to probe a machine to check how it reacts in a very specific situation. From the sometimes incomplete information gathered, one has to deduce the OS running on the machine. An example of a passive deduction test is presented in Example 3.2.

Example 3.2 (passive test ARP Request (Test-2))
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 B.2. ♢

3.3.1 Passive Tools

A few examples of passive tools for OSD are: SinFP [3], p0f [80], Siphon [68], and ettercap [50]. 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.

3.3.1.1 p0f

We consider p0f version 2.0.8 here. p0f is one of the most popular passive OSD tool available. It offers four operating modes: Syn, SynAck, RstAck and StrayAck; the

latter is an experimental mode. Each mode consider only a single type of traffic. The Syn mode considers only SYN packets sent by the target; and this corresponds to Test-1 of Definition B.1. The SynAck (resp. RstAck) mode only analyzes how the target responds to a SYN packet sent to an open (resp. closed) port; and this corresponds to Test-8 (resp. Test-9) of Definition B.8 (resp. B.9). Finally, the StrayAck mode considers the TCP packets exchanged during the session (as opposed to during the initialization of the session), we have no corresponding test for that.

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

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

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 several 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 up 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 were the only packet available. This has two major implications. First, p0f is stateless; for each packet it will guess the OS without considering the packet previously seen. 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.

Finally, the p0f engine is a simple string matching algorithm. Figure 3.1 provides a small peek at the p0f signature file for the Syn mode. Each entry is of the form

$$W : T : D : S : O : Q : OS : Details \tag{3.1}$$

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, 2.2 for Linux).

Each SYN packet is transformed to a string representation (the first 6 fields of Equation 3.1) and then matched against the signature file. p0f returns the OS contained in the first entry that matches with the packet, thus every OS having the same behavior must be included in the same entry (see first entry in Figure 3.1). 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? Either FreeBSD or NetBSD (see last entry in Figure 3.1 for a workaround).

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 to adjust the TTL when the fingerprinted host is on a different network (the packet has to go through a router).

Figure 3.1: p0f Signature File Format

```
%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)
```

- 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.

- Separate signature files. p0f has a signature file for each operation mode and the signatures can be updated quite easily without modifying the engine. However, due to the basic rule-matching algorithm, it is hard to know how p0f will handle new entries in the signature file.

3.3.2 Advantages and Limitations

The principal advantage of the passive approach is its non-intrusive nature: a passive tool does not need to inject anything on 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[3]. These limitations are all part of the reason why passive tools do not achieve better performance in IDS context gathering, see Chapter 2.

3.3.2.1 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

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

that less information can be deduced from usual (well-formed) communication than from carefully engineered (and possibly malformed) stimulus-response sequences.

3.3.2.2 Restriction to Single-Packet Rules

Every passive OSD tool that we are aware of uses 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 by a single packet. This is the case with the ARP Request Retransmission test (or any retransmission test for that matter) where the goal is to monitor the delay between retransmissions of the same request, see Definition B.6.

Second, many phenomena can be represented by a single packet but at the cost of losing precious information. This is the case of stimulus-response tests. For instance, sending a TCP Syn packet to an open port will trigger a TCP SynAck as an answer, see Definition B.8. It is possible to simply analyze the response packet and deduce some information; however, some field of a SynAck can be influenced by the value of a field in the original Syn packet (e.g. it is not uncommon that the DF bit in the SynAck packet takes the exact same value as the DF bit in the corresponding Syn packet).

3.3.2.3 Lack of Memory

Finally, all existing passive tools are stateless, i.e., they do not have a memory. For each packet they analyze, they provide the set of possible OSes for 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 limiting 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.

3.4 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 a TTL in the SYN/ACK), see also Example 3.3. Another possibility is to stimulate the target with a malformed packet and analyze how it will behave, see Example 3.4. Note that the malformed packets discussed here are not malicious and thus totally harmless to the target.

Example 3.3 (active test with well-formed traffic)
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 correspond to Test-9 of Definition B.9. ♢

Example 3.4 (active test with malformed traffic)
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 ignores the bogus packet. This corresponds to Test-22 of Definition B.17. ♢

Another kind of active tests consists in placing the target machine in extreme conditions and monitoring how it behaves. However, putting a machine in extreme

conditions often results in disrupting normal activities. [79] 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 may provide information concerning the OS. Since these tests are disruptive, we will not consider them here.

3.4.1 Active Tools

Nmap [78] and Xprobe [2] 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.

3.4.1.1 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 and we consider version 4.75.

Nmap considers TCP and ICMP packets. It relies on the following 11 tests (see Appendix B): 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.

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 always 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 many 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 on 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 tests requiring malformed packets (Test-21 and Test-22). Thus it will inject at least two, but sometimes up to 40, malformed packets.

Figure 3.2 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 3.2. 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 T1 corresponds to Test-20 (see Definition B.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 handles duplicate entries (unlike p0f).

Figure 3.2: Nmap Signature File Format

```
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)
```

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 two fundamentally distinct behaviors with respect to a test[4], 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.

- 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 (and even in deciding if the target OS is vulnerable or not to a given attack). This can, at least partially, explain the lack of accuracy for Nmap in the experiment of Chapter 2.

Finally, from the structure of the signature file we can understand why Nmap always executes all of its tests. In order for a signature to match (and thus for Nmap to provide an answer to the user), all the requirements of that signature must be observed. The only way to make this possible is to perform all the 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 results and signatures are less likely to match.

3.4.1.2 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 (thus Xprobe does not vary the stimulus field values), except Test-11 which is executed twice (with different TOS, DF and ICMP code vales) and Test-12 (the only malformed test used by Xprobe) which is almost never used (it seems like this specific test must be requested by the user, although Xprobe's documentation affirms it should always be performed).

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

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 loosing too much accuracy, the case where the two ports tried have the same status.

The rule format of Xprobe (see Figure 3.3) is similar, 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 3.3 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 to yes and uses a TTL no greater than 128 (actually it uses a TTL of exactly 128, but this value is decreased by routers) for that test. Xprobe can handle two identical entries (two different OSes with the exact same signature) without any problems.

Figure 3.3: Xprobe Signature File Format

```
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
...
}
```

From the signature of Figure 3.3, 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. A signature can be added and 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) [39] 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 associates to each signature a score for each individual test (based on the results obtained from the scan). It then computes the score of each signature by summing the signature score for each test. It finally selects the signature(s) with the highest score and outputs the corresponding operating system(s). This fuzzy matching helps Xprobe avoid a costly port scan.

3.4.2 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 they usually relies on malformed packets.

3.4.2.1 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, most active tools need to know about one open and/or one closed port on the target to perform the tests. Many of those tools, such as Nmap, perform a port scan before doing their tests. A port scan by itself can sometimes generate more than a thousand network packets. Some tools offer an option to disable the port scan, however the accuracy dramatically drops in that case. In theory, with a 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, which means the total number of packets will be sent every time. 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 like "Is a given machine running the operating system θ?", an active tool would have to reason to generate a judicious sequence of actions; a feature that is not available in today's active tools.

Finally, another problem with using active tools in the context of IDS 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 not acceptable in the context of IDS since we expect the same query to be repeated many times and we do not want to generate too much traffic.

3.4.2.2 Malformed Traffic

Another major drawback of active tools is the injection of malformed packets on the network. The usage of malformed packets is justified by the fact that usually one can glean more information from the response to an 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, responding with an error message, not responding at all, etc.). This malformed traffic becomes a huge problem when it interferes with other network equipments, for instance an intrusion detection system. By using planning, we could try to avoid as much as possible the injection of malformed packet.

3.5 Testing Current Tools for OSD

In Chapter 2 we evaluated how current OSD tools perform for the task of IDS context gathering. However, most OSD tools where not designed for that specific situation; they were developed to find out the actual operating system running on a given computer. In this section, we consider the more classical OSD task (finding out the actual OS) and evaluate current OSD tools with respect to that task.

The setting consists of 84 computers: 83 (the targets), each with its own OS, targeted for OS discovery and 1 (the actor) used to initiate communication sessions with the targets. We gathered 6,656 traffic traces each containing between 1 and 63,000 packets. Each trace corresponds to a communication session between the actor and one target. These traces contain attack traffic as well as more classical traffic such as connection requests, file transfer, telnet sessions, etc.

In the experiment, we tested seven passive OSD tools (Siphon 0.666 [68], the Syn, SynAck, RstAck, and StrayAck modes of p0f 2.0.8 [80], SinFP 2.00-8 [3], ettercap NG-0.7.3 [50]) as well as two active tools (Nmap 4.20 [78] and Xprobe 2.0.3 [2]).

We fed each trace to each passive tool and cumulated the tool's guesses over that trace to form the set of possible OSes. On the other hand we ran each active tool directly against the machines targeted for OS discovery and recorded the result. We then use this result to be the set of possible OSes for that tool with respect to any given trace. Based on that, we say that a tool returns:

- the correct answer when the set of possible OSes returned by the tool contains the actual OS.
- the incorrect answer when the set of possible OSes returned by the tool does not contain the actual OS.
- unknown when the set of possible OSes returned by the tool is empty.

We believe it is better for a tool to return unknown than to return the incorrect answer.

In this experiment, we focus on two measures:

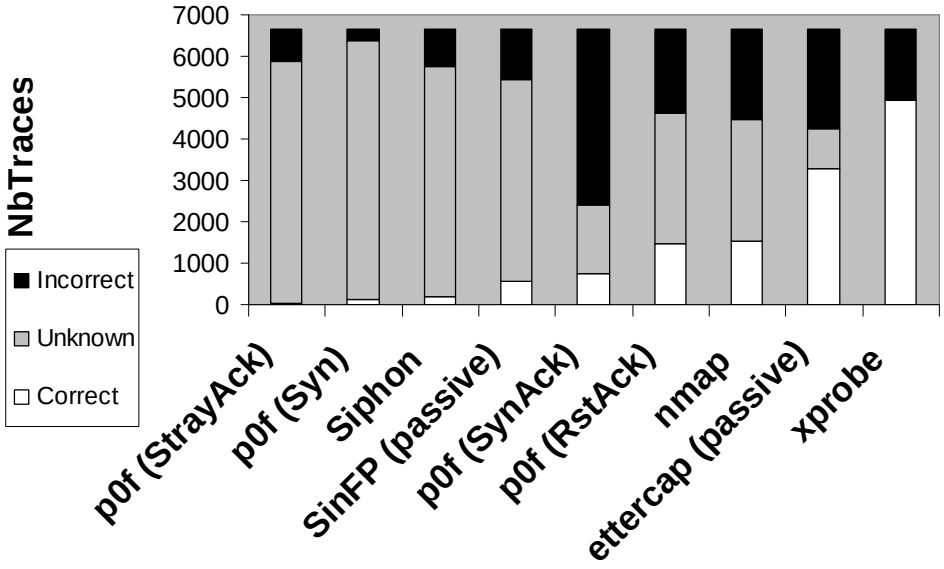


Figure 3.4: Recall for OSD Tools

- Recall: the number of traces for which the tool provided the correct answer vs the number of traces for which tool provided the incorrect answer or unknown.
- Precision: for each trace for which the tool provided the correct answer, what was the size of the possible OSes set returned.

3.5.1 Recall

The results, shown in Figure 3.4 (Table 3.2 presents the actual numbers), give for each tool: the number of traces for which the tool answers correctly (the white area); the number of traces for which the tool answers unknown (the grey area); and the number of traces for which the tool answers incorrectly (the black area).

We can extract several interesting things from these results:

- It is surprising to see how poorly most of the tools perform; especially Nmap and p0f which are widely used. Every tool, except the two best ones, have a recall below 25%. Two tools clearly stand out as being rather good: Xprobe and ettercap.
- The two best tools provide the incorrect answer in 25% of the cases or more. This makes them quite unreliable.

Table 3.2: Recall for OSD Tools

Measure / Tool	Correct Answers	Unknown Answers	Incorrect Answers
p0f StrayAck	28 (0.42%)	5847 (87.85%)	781 (11.73%)
p0f Syn	135 (2.03%)	6244 (93.81%)	277 (4.16%)
Siphon	183 (2.75%)	5574 (83.74%)	899 (13.51%)
SinFP	562 (8.44%)	4878 (73.29%)	1216 (18.27%)
p0f SynAck	746 (11.21%)	1661 (24.95%)	4249 (63.84%)
p0f RstAck	1461 (21.95%)	3159 (47.46%)	2036 (30.59%)
Nmap	1534 (23.05%)	2922 (43.90%)	2200 (33.05%)
ettercap	3294 (49.49%)	955 (14.35%)	2407 (36.16%)
Xprobe	4933 (74.11%)	0 (0.00%)	1723 (25.89%)

- In the experiment of Chapter 2, three tools were relatively close in the lead, ettercap (passive), p0f (SynAck), and Xprobe in order. Here, p0f (SynAck) is not even close while the other two are still in the lead.

- In the experiment of Chapter 2, the advantage of the active approach was not very clear, since the best passive tool was almost as good as the best active tool. Here, the advantage is clearly seen as the best active tool, Xprobe, is much better than the best passive tool, ettercap.

- the number of incorrect answers seems to increase proportionally with the number of correct answers. This is not a surprise as the more aggressively an approach guesses answers, the more likely it is to make mistakes.

3.5.2 Precision

The results presented above were directed toward recall, that is the ability of a tool to include the actual OS in its set of possible OSes. It is easy for a tool to obtain an arbitrarily good recall, by including every single OS in its set of possible OSes. However, such a tool would not be helpful as it would not provide any information to the user. To circumvent this problem, we also consider precision, that is the ability of a tool to provide the smallest set of possible OSes. Thus we look at the size of the possible OSes sets returned by the tools during the experiment.

Table 3.3: Precision Summary for OSD Tools

Measure / Tool	Average Size of Set of Possible OSes
p0f StrayAck	6.57
p0f Syn	36.85
Siphon	10.36
SinFP	6.69
p0f SynAck	4.46
p0f RstAck	18.58
Nmap	4.86
ettercap	21.37
Xprobe	12.30

Table 3.3 provides, for each tool, the average size of its set of possible OSes for all the cases where it returned the correct answer. Actual sizes of set of possible OSes are provided in Section C of Appendix C.1.

We consider only the tools that have a "good" recall; that is, we consider Nmap, p0f (RstAck), ettercap, and Xprobe. Nmap and p0f (RstAck) both have a recall around 25%. However, their precisions are quite different. Nmap has an average size of 4.9 while p0f is at 18.6. The difference is partially explainable by the fact that Nmap is active while p0f is passive. Xprobe has a significantly better recall than ettercap. The same occurs with precision: average size of 12.3 for Xprobe and 21.4 for ettercap.

The extremely good precision for Nmap could be an explanation for its poor recall. Nmap might have been designed to provide an answer only when it can narrow it down to less than 10 OSes or so.

3.5.3 Discussion

To summarize the results provided by this new experiment, we can state that current OSD tools are either not very good or not too reliable. Even Xprobe, which clearly stands out as the best tool, still guesses incorrectly once out of four times.

From the point of view of precision, Xprobe seems relatively good with an average number of guessed OSes around 12. It is not clear what we can expect from the

precision point of view, as many versions of the same OS (e.g., Windows 2000 sp1 and Windows 2000 sp2) behave in a very similar way; distinguishing between them might not always be possible. An interesting experiment would be to compute the clusters[5] of OSes from the point of view of each tool. However, the lack of knowledge representation of current tools do not help such a task.

3.6 Advantages and Limitations of the Current Approaches and Tools

The passive approach has mainly two advantages:

- It is non-intrusive as it does not send anything on the network.
- It has access to information that cannot be obtained actively (using a stimulus). Test-1 and Test-26 are two such examples.

On the other hand, the passive approach has an important limitation:

- Communication cannot be requested and the tool must wait for somebody to initiate it. Thus, the information may not be available when needed.

Current passive tools have two other drawbacks related to their implementation (not intrinsic to the passive approach):

- Only single-packet rules are used. This prevents the tool from observing artifacts that are dependant on the stimulus (in stimulus-response tests) or requires several packets (in sample packet tests). The result is a loss of accuracy.
- Passive tools are stateless. They always only consider only the current packet and don't take advantage of the information extracted from the previous packets. This results in a loss of accuracy and a divergent set of guessed OSes. The divergence of the set of possible OSes is problematic for automatic decision making, such as IDS alarm filtering. Since another OS can always be added to the current set of possible OSes, we can never assume that some OSes are impossible and act based on this information

[5]Two OSes being in the same cluster if the tool cannot distinguish them.

The active approach has two important advantages:

- It can stimulate the host to obtain (missing) information at anytime. Moreover, it can reuse the same stimulus with different values to obtain more information. For instance, by sending two TCP SYN packets to an open port, one with the DF bit set and the other with the DF bit not set, it is possible to classify the OS in one of three categories: "responds with DF set", "responds with DF not set", or "responds with same DF as stimulus".

- The focus on the sole objective of always obtaining the actual OS results in a lack of flexibility. It makes it harder to incorporate an OSD tool in a specific application (an IDS for instance).

- It can use specially crafted (malformed) stimuli to see how the target will react in that specific situation. Usually, more information can be extracted from unusual situations.

Unfortunately, active tools have a major drawbacks: they are intrusive. There are several reasons why active tools are intrusive:

- They generate a lot of network traffic:

  - They always run all their tests before analyzing the results. This often leads to the execution of tests that have no discriminant power in the current situation.

  - Some tools must perform a port scan before sending their fingerprinting stimulus, otherwise their accuracy drops dramatically.

  - Each time the user queries an active tool, the tool must perform its tests. It cannot rely on previous runs as the information might now be outdated.

- Some tools rely on malformed traffic. This is problematic when that traffic interferes with other network devices (e.g., IDSes).

3.7 Obfuscating OS Fingerprints

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 [33]: 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[6] and difficult. Moreover, they must be applied individually to every host on the network. For those reasons, it is recommended [33] not to adopt a host-based OS fingerprint obfuscation method. A host-based modification would disrupt the ability to use OS information as IDS context.

The network-based approach [66] consists of adding a traffic scrubbed device at the single entry point of a network. This device modify the egress traffic making every computer in the network look identical, thus preventing OS discovery, form outside. The main advantage of this technique is that it handles every computer in the network at once. Moreover, it does not prevent gathering OS information to use as IDS context, because the traffic inside the network remains unchanged and exhibit the difference in TCP/IP stack implementations.

In conclusion, host-based techniques to obfuscation OS fingerprints would prevent gathering target information for IDS context, but their use is proscribed (both for their difficulty to deploy and the risk they incur). On the other hand, network-based obfuscation techniques still allow OS discovery from inside the network (enabling OS information gathering for IDS context) and are much safer, easier, and faster to deploy.

[6] According to [33], modifying the stack implementation could prevent some application from running correctly.

3.8 Conclusion

In the previous chapter, we established, among other things, that current tools for OSD are not adequate for gathering IDS context information.

In this chapter, we studied the two classical approaches (active and passive) to OS discovery. Both have advantages as well as drawbacks. Of particular interest are the following:

- Passive tools are stateless and they can only get information from the packets already on the network. This clearly makes them unsuitable for IDS context gathering.
- Active tools are intrusive and thus they only consider a very small set of tests (to avoid flooding the network with stimuli) and they sometimes rely on malformed traffic. The former limits their accuracy and the latter makes them inappropriate for coexisting with IDSes.
- All existing tools lack the ability to continuously monitor the network in order to converge toward a better knowledge state. This is exactly what is needed in the case of IDS context gathering.

In the light of these drawbacks, we propose, in the next chapter, a new hybrid approach to OS discovery. Our hybrid approach combines the advantages of both approaches and addresses their limitations.

Chapter 4

Problem Statement

Knowing where you're going is all you need to get there.
-Frederick Frieseke

Based on the experiments of Chapter 2, we know that the target configuration is an important piece of information for filtering non-critical IDS alarms. We also know that current OSD tools are not adequate to gather such information.

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.

Section 4.1 provides more information about these objectives and their importance.

Research toward these three objectives is well underway and we already have some interesting results: a prototype for our new tool, a theoretical model for our approach and the framework of an environment to collect OS fingerprints. These early results are briefly discussed in Section 4.2 and in more details in Chapter 5 to 8.

There is still work to do before fully achieving the three objectives mentioned above. The work we propose to do with respect to each of the above objectives is described in details Chapters 5 to 8 and summarized in Chapter 9.

4.1 Objectives

Here we describe the three objectives in more details and we argue why they are important.

4.1.1 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 really needs.

We believe these improvements over other existing tools will make our tool very attractive and usable both by researchers, network administrators, and security administrator. Our tool should also be useable by a third party tool, which is difficult with current tools due to their ad hoc and informal flavor.

4.1.2 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 interesting insights regarding the complexity of our problem.
- It will provide us with well-established algorithms to implement the different modules.
- Improvement in the general theory will automatically enhance the tool.
- It will serve as an elegant description of the tool engine.

4.1.3 Systematic Collection of OS Fingerprints

One of the problems underlying the OS discovery field is the gathering of OS fingerprints. Fingerprints are used for distinguishing OSes in different situation. Usually, they take the form of rules (as seen in Chapter 3). Since several new OS versions are released each year, OSD tools must constantly be updated with the new fingerprints. The problem is to obtain those fingerprints. 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 automatic way. Thus it will be easy to obtain fingerprints for new OSes and since the experiments can be reproduced, we can be sure the fingerprints are correct.

4.2 Early Results

Here we briefly discuss the early results achieved so far with respect to each objective. More details can be found in Chapter 5 to 8

4.2.1 A Better Tool

Our main idea to obtain a better OSD tool is to combine the active and passive approaches into a new hybrid approach guided by knowledge management. We have developed a proof-of-concept tool using answer set programming. We have experimented with the passive module and results are extremely promising. We already see an improvement over other passive tools and we are already close to the performance of the best tools (mainly due to the knowledge-oriented approach). The prototype can be downloaded from [11].

More details about the new hybrid approach and its first implementation as well as the experiment results can be found in Chapter 5.

4.2.2 A Strong Theoretical Background

The prototype discussed above, described in details in Chapter 5, was developed to be a proof-of-concept to see whether or not the knowledge-oriented approach would provide an interesting improvement. However, the choice of Answer Set Programming as the implementation technique pretty much guaranteed that the tool would be inefficient. Once we were convinced that our idea of an hybrid approach to OSD would be interesting, we started to look at the complexity of the problem in order to design an efficient algorithm. We found that the theory of diagnosis problem solving [60] would be a good model for our problem.

In terms of diagnosis, the passive module corresponds to *candidate generation* while the active module corresponds to *candidate elimination*. Looking at the algorithms available for candidate generation, and based on the specific setting of our OSD problem, we found a (low order) polynomial algorithm that would solve the passive module of our hybrid approach. For the active module, the diagnosis theory of candidate elimination also provides a solution. However, the solution is not flexible enough for our needs as it does not allow the user to direct the process by specifying a goal (e.g., to know if the target is vulnerable to a specific attack).

More details about representing OSD as a diagnosis task will be presented in Chapter 6 and 7.

4.2.3 Systematic Collection of OS Fingerprints

Our third objective is to provide a systematic and automated way of gathering OS fingerprints.

As mentioned in Chapter 3, tests are either passive, active or both. Tests that are passive only are based on spontaneous events, i.e., events that must be initiated by the target computer and cannot be triggered remotely on-demand. Other tests are based on reactive events, i.e., events occurring in response to a stimulus. As a consequence, the fingerprints are either based on spontaneous or reactive events.

Using the virtual network environment available at CRC (described in Section 8.3.1 of Chapter 8), we can automatically gather fingerprints based on reactive events

(we only need to provide an adequate stimulus and capture the response). However, this environment does not allow to automatically collect fingerprints based on spontaneous events (as they cannot be triggered by a stimulus).

To circumvent this limitation, we are building a tool, called VNEC (Virtual Network Experiment Controller) to control a virtual environment. The tool lets the user specify a *network experiment* and then executes the experiment on a set of virtual machines. This tool provides the user with a way to dispatch commands to any virtual machine. Thus, it can be used, for instance, to ask each target OS to initiate a TCP connection, providing the SYN packet we need to extract the SYN fingerprint for that target. However, it is general enough that it can be used for other purposes than OS fingerprinting. For instance, it can be used to perform security sensitive experiments such as studying attack reaction behaviors and virus spreading patterns. More details about the tool can be found in Chapter 8. A first prototype of this tool is available from [12].

Chapter 5

HOSD - A Hybrid Approach to Operating System Discovery

Before a man attempts to fly, to walk upright he should try.
-Unknown Source

Abstract

In the previous chapters, we established the importance of target configuration as contextual information for IDS and we established that current OSD tools are not adequate to gather this information. Moreover, we identified the limitations of the current OSD approaches and tools. In this chapter, we propose a new hybrid approach to OSD; we call it HOSD (Hybrid Operating System Discovery). This approach combines the advantages of the passive and active approaches while being much more flexible. We also discuss a prototype implementation of HOSD using answer set programming. Finally, we use our implementation to compare the effectiveness of our new OSD approach against existing tools.

5.1 Introduction

In the previous chapters, we have seen that current OSD tools do not provide very good results in practice. By studying the classical approaches (active and passive), we have established their main limitations. For instance, the absence of memory, the lack of reasoning capabilities, and the noisiness.

Here we propose a hybrid approach to operating system discovery that we call HOSD (Hybrid OS Discovery). HOSD fixes many limitations of the classical approaches and keeps most of their advantages.

The rest of this chapter is structured as follows. First, Section 5.2 introduces the concept of hybrid operating system discovery. Section 5.3 describes an implementation using answer set programming. Then Section 5.4 illustrates the effectiveness of our hybrid approach by comparing HOSD with existing tools using the same experiments discussed in the previous chapters. The chapter will end with a short conclusion, a summary of the contributions made throughout this chapter, and the

proposed work during the thesis.

5.2 The Hybrid Approach

This section presents a hybrid framework to operating system discovery (OSD) that encompasses the advantages of the two classic approaches (active and passive) while being more flexible. Among other things, HOSD 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 tools, as it has a memory and can rely on active tests when needed.

Moreover, we will see how the two modules (active and passive) interact together to fulfill the user's needs.

5.2.1 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 5.1 shows a situation when a hybrid tool goes into active mode.

Example 5.1 (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 many advantages over the active and a passive ones. 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 unusual 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 precision (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.

5.2.2 The Passive Module

The passive module has to fulfill two important requirements:

- To continuously monitor the network.
- To 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. At the beginning, the knowledge based 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 base method differs from the guess-based method of current OSD tools (As seen in Chapter 3). The most important difference is the way we can interpret the current set of possible OSes. With current passive tools, since the current set of possible OSes P is made up of an aggregation of several guesses, we cannot say that if $o \notin P$ then o is not the actual OS. In fact, new information can come in for which the tool would provide o as a guess. With our approach, whenever $o \notin P$ we can conclude that o is not the actual OS.

5.2.2.1 Querying the Knowledge Base

Manipulating a knowledge base gives us the flexibility of querying it in different ways. We believe the following queries to be particularly relevant for our OSD application:

Single OS Query: is the computer running the specific OS o?

Group OS Query: is the computer running an OS in the given set O?

Exact OS Query: which OS is running on the computer?

The Single OS Query[1] could be useful to determine which computers need a specific update (e.g., if Windows XP sp2 needs to be updated to Windows XP sp3). The Group OS Query is definitely useful for determining if a computer is vulnerable to a specific attack (here O would be the set of vulnerable OSes). Finally, the Exact OS Query is the usual OSD query and is obviously useful for gathering general information about a network.

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

5.2.3 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. For classical active OSD tools, the information from several tests is correlated to construct the set of possible OSes. 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:

[1]The Single OS Query is a special case of the Group OS Query (where O is a singleton). The real importance of the Single OS Query will come later as we believe it is (computationally) easier to solve in some cases.

- 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[2]. 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).

5.3 A First Implementation

In order to validate our initial intuitions that the hybrid approach (mainly the knowledge-oriented aspect) will provide better results than the classical approaches, we implemented a proof-of-concept tool.

This section first discusses the requirements with respect to the knowledge management aspects of our approach. Then it introduces *answer set programming* as a language satisfying those requirements. Finally, it provides a description of the implementation of the passive and active modules.

5.3.1 Knowledge Representation Language

When building an application that relies extensively on the management of a knowledge base, it is important to choose the underlying language wisely. In the context of hybrid OS discovery, we want a language that meets the following requirements:

[2]Active OSD tools, however, always try to find out the exact OS

- Is declarative.
- Supports a *weak form* of non-monotonic reasoning.
- Can be used for knowledge management and reasoning.
- Has a sound and complete semantic.

Having a *declarative* language minimizes the effort needed to update the program with new scenarios, i.e. new signatures for new operating systems. It also opens the door to automatic generation of the program (from the database of fingerprints). Since we want to compute the set of possible OSes given some information, and then we want further information to refute some of these possible OSes, the language must support a *weak form of non-monotonic reasoning* (that is the ability to draw conclusions that can be retracted as soon as more information becomes available). From a given set of network packets it should produce the set of all possible OS so far, and as soon as a new packet comes in we expect the set of possible OS to decrease as much as possible, thus retracting some previously made conclusions. The language should knowledge management, allowing querying of the knowledge, and reasoning, for test selection. Finally, a sound and complete semantics will ensure that the results returned are exactly the results expected.

A quick glance at these requirements should trigger the idea of using a language like Prolog for our task. Albeit Prolog is widely used for knowledge-based applications, it has a few limitations for OS discovery purpose. First, Prolog is in principle a declarative language, but it has some procedural elements that make its semantics somewhat unclear. Furthermore, due to its top-down and depth-first evaluation strategy, the ordering of the rules (and the ordering of the terms inside a rule) is significant in Prolog; thus, this could greatly restrict the possibility of automatically generating the program. Also, Prolog does not support non-monotonic reasoning[3]. Finally, Prolog does not have a complete semantic. Thus the set of answers given by Prolog is not always the set of all intended answers.

As an alternative to Prolog, we chose to use another logic programming language paradigm, called Answer Set Programming (ASP). Let us give a brief introduction

[3]Nor the weak form we need.

to ASP and then it will be possible to show why ASP is appropriate for hybrid OS discovery.

5.3.2 Answer Set Programming

As an alternative to Prolog, we chose to use disjunctive logic programming under the answer set semantics (a.k.a Answer Set Programming or ASP). Below, we give a brief introduction to ASP and then show why it is appropriate for HOSD.

ASP consists of declarative programming using extended disjunctive logic programs with an answer set semantics. It admits rules of the form

$$L_1 \vee \cdots \vee L_k \leftarrow L_{k+1}, \cdots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$

where each L_j is a classic literal, i.e. an atom A or its *classic negation* $\neg A$, and *not* denotes *weak negation*[4]. We call $\{L_1, \cdots L_k\}$ the head of the rule, while $\{L_{k+1}, \cdots L_m, not\ L_{m+1}, \ldots, not\ L_n\}$ forms its body. We can already see two major extensions to Prolog:

- we can have disjunctions in the head of rules. A rule like $Q(a) \vee S(a) \leftarrow P(a)$ means that if $P(a)$ is true, then at least one of $Q(a)$ or $S(a)$ must also be true;
- we have both classic (or strong) negation and weak negation (or negation as failure), see Footnote 4.

This additional expressiveness of ASP will be particularly useful in our case. For instance, a rule describing that a TCP SYN packet with the DF bit set and a TTL of 64 must originate from a machine running Windows 2000 or Windows XP could be expressed as follows:

$$os(I, win2000) \vee os(I, winXP) \leftarrow tcp(I, \_, \_, \_, yes, syn, 64).$$

Example 5.2 presents a valid ASP rule together with its meaning.

$$P(x) \vee \neg Q(x) \leftarrow R(x), \neg S(y), not\ T(x), not\ \neg U(x) \tag{5.1}$$

[4] $\neg P(a)$ is true iff $P(a)$ is false, while *not* $P(a)$ is true if we can consistently assume $P(a)$ to be false.

Table 5.1: The ground program of rule (5.1)

$R(a)$		
$S(b)$		
$P(a) \vee \neg Q(a)$	$\leftarrow$	$R(a), \neg S(a), not\ T(a), not\ \neg U(a)$
$P(a) \vee \neg Q(a)$	$\leftarrow$	$R(a), \neg S(b), not\ T(a), not\ \neg U(a)$
$P(b) \vee \neg Q(b)$	$\leftarrow$	$R(b), \neg S(a), not\ T(b), not\ \neg U(b)$
$P(b) \vee \neg Q(b)$	$\leftarrow$	$R(b), \neg S(b), not\ T(b), not\ \neg U(b)$

Table 5.2: A simple program

$R(a)$		
$P(b)$		
$P(x) \vee Q(x)$	$\leftarrow$	$R(x)$

Example 5.2 (rules)

The rule of Equation 5.1 is valid and means that if $R(x)$ is true, $S(y)$ is false, $T(x)$ can be assumed to be false, and $U(x)$ can be assumed to be true, then $P(x)$ is true or $Q(x)$ is false. We may consider the program to be *ground*, i.e. all its variables are instantiated in the underlying domain of the program (its Herbrand universe [4]). For instance, the ground program formed by the rule of Equation (5.1), together with the facts $R(a)$ and $S(b)$ (where a and b are constants), is shown in Table 5.1. $\diamondsuit$

Definition 5.1 (Answer Set)

A program is a set of rules of the form introduced above. An answer set S of a program Π consists of a particular set of ground literals such that:

- *the literals of S are those made true by Π;*
- *the literals of S are sufficient to respect the constraints of Π's rules;*
- *no proper subset of S is also an answer set of Π.* $\bigcirc$

A program may have zero, one or many answer sets.

Example 5.3 (answer sets)

Consider the program in Table 5.2. The answer sets of this program are $\{R(a), P(b), P(a)\}$ and $\{R(a), P(b), Q(a)\}$. Note that $\{R(a), P(b), P(a), Q(a)\}$ is not an answer set, as

it is not minimal (we can remove either $P(a)$ or $Q(a)$ and still satisfy the constraints defined by the program rules). $\{R(a), P(b)\}$ is not an answer set, as it does not satisfy the constraint induced by the rule $P(x) \vee Q(x) \leftarrow R(x)$ (because $R(a)$ is true, one of $P(a)$ or $Q(a)$ must also be true). $\diamondsuit$

As we can see, ASP is a declarative programming language, but unlike Prolog, which uses top-down and depth-first search, the evaluation strategy is not fixed in ASP. Moreover, its set-based semantics ensures that the ordering of the rules is not important. Thus, automatically generating an ASP program seems possible. ASP has a clear declarative semantics that Prolog lacks. Moreover, in ASP it is possible to specify constraints to prune undesirable models and customize the inference engine. A constraint is a rule with an empty head that prevents the body from ever being true. A weak constraint also has an empty head and tries to prevent, as much as possible, the body from being true. If it is not possible to find a model without violating any weak constraint, the models that violate as few weak constraints as possible are returned. ASP fully supports non-monotonic reasoning, see Example 5.4.

Example 5.4 (non-monotonicity of ASP)
Consider the program formed by the single rule

$$a \leftarrow p \wedge not\ q$$

together with the single fact p. At this point, a is considered true as supported by the unique answer set $\{a, p\}$. However, if we add the fact q, then a is not true anymore (the only answer set is now $\{p, q\}$). $\diamondsuit$

Many problems of a combinatorial nature can be represented and solved under this programming paradigm [4]. ASP has also been used for planning/reasoning purposes ([30], [32], [73], and [42]). The interested reader is referred to [4], [74], and [37] for a deeper presentation of ASP. We use DLV [23] as the engine for evaluating our ASP programs.

5.3.3 Implementing the Passive Module using ASP

The passive OS discovery module is essentially a knowledge base; it is updated for every new captured network packet and queried whenever information is needed about

the operating system of a given machine. Section 5.3.3.1 presents the Intentional DataBase (IDB) file containing ASP rules that model the behavior of different operating systems; Section 5.3.3.2 presents the Extensional DataBase (EDB) file containing the recorded network packets and explains the querying process; and Section 5.3.3.3 discusses the kind of queries that are supported by our passive module vs other passive tools.

5.3.3.1 Passive IDB

Figure 5.1 presents a fragment of our actual IDB file for passive OS discovery (*passiveOSfingerprinting.IDB*) where ":-" stands for right to left implication "←". The first rule is a weak constraint stating that unless necessary, one machine should not be assigned two different operating systems in a single answer set (each answer set will correspond to a *possible* assignment of OSes for the computers).

The set of rules in the **ARP Request** group models different behaviors of an OS regarding the destination MAC address in an ARP request (as explained in Example 3.2 of Chapter 3). The predicate $arp(X, Y, Type, Mac)$ represents an ARP packet sent from X to Y with a given message type $Type$ (request, reply) and where Mac contains the destination MAC address.

The second set of rules, those in the **TCP Syn** group, represents different possible behaviors for the sender of the first packet of a TCP handshake. The predicate $tcp(X, Y, Xport, Yport, DF, Flags, TTL)$ represents a TCP packet sent from X through port $Xport$ to Y on port $Yport$ where DF indicates whether or not the DF bit is set, $Flags$ lists the TCP flags that are set (SYN, ACK, RST, ...), and TTL contains the time-to-live value.

The last rules presented here, those in the **TCP SynAck** group, capture the possible behaviors for the sender of the second packet of a TCP handshake (with respect to its SYN stimulus). Here, again, the value of the TTL and the DF bit are monitored.

```
%One IP should not correspond to two different OSes
:∼ os(X,Y), os(X,Z), Z != Y.
%ARP Request
os(X,linuxRH7_1) ∨ os(X,linuxRH5_2) ∨ os(X,linuxRH8_0) ∨
   os(X,winXP) ∨ os(X,win2k):- arp(X,_,1,mac00_00_00_00_00_00).
os(X,macOS) ∨ os(X,sunOS) :-
  arp(X,_,1,macFF_FF_FF_FF_FF_FF).
os(X,freeBSD5_0) :- arp(X,_,1,Y), Y != mac00_00_00_00_00_00,
   Y != macFF_FF_FF_FF_FF_FF.
%TCP Syn
os(X,linuxRH5_2) :- tcp(X,_,_,_,no,syn,64).
os(X,win2k) ∨ os(X,winXP) :- tcp(X,_,_,_,yes,syn,128).
os(X,sunOS) ∨ os(X,linuxRH7_1) ∨ os(X,macOS) ∨
  os(X,linuxRH8_0) ∨ os(X,freeBSD5_0) :- tcp(X,_,_,_,yes,syn,64).
%TCP SynAck
os(X,linuxRH5_2) :-
  tcp(Y,X,Yport,Xport,_,syn,_),
  tcp(X,Y,Xport,Yport,no,syn_ack,64).
os(X,win2k) ∨ os(X,winXP) :-
  tcp(Y,X,Yport,Xport,_,syn,_),
  tcp(X,Y,Xport,Yport,yes,syn_ack,128).
os(X,macOS) ∨ os(X,sunOS) :-
  tcp(Y,X,Yport,Xport,_,syn,_),
  tcp(X,Y,Xport,Yport,yes,syn_ack,255).
os(X,freeBSD5_0) ∨ os(X,linuxRH7_1) ∨ os(X,linuxRH8_0) :-
  tcp(Y,X,Yport,Xport,_,syn,_),
  tcp(X,Y,Xport,Yport,yes,syn_ack,64).
```

Figure 5.1: Some Rules for Passive OS discovery

5.3.3.2 Passive EDB

Every time a network packet is captured, it must be added to the knowledge base. This is done by adding a fact representing the packet to the EDB file (*passiveOSdiscovery.EDB*). Below are two examples explaining how passive OS discovery works in ASP.

Example 5.5 (a first packet)

Suppose the only packet captured so far is an ARP request from machine 10.1.1.6

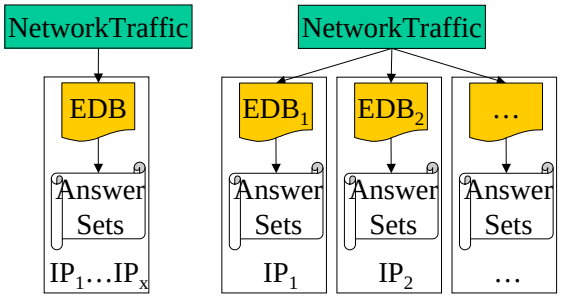
to machine 10.1.1.1 where the destination MAC field is filled with zeroes; this is represented, in the EDB file, by the fact $arp(ip10\_1\_1\_6, ip10\_1\_1\_1, 1, mac00\_00\_00\_00\_00\_00)$. DLV computes all possible answer sets and comes up with five. Each answer set contains one possible operating system for 10.1.1.6. The first rule of the **ARP Request** group (see Figure 5.1) is used and one of the terms in the head has to be true. That is, the OS for 10.1.1.6 is one of Windows 2000, Windows XP, Linux Red Hat 5.2, Linux Red Hat 7.0 or 8.0. ♢

Example 5.6 (a second packet)
Now suppose that a second packet is captured. This second packet is a TCP SYN packet from 10.1.1.6 to 10.1.1.1 with the DF bit set and a TTL of 64. So, the knowledge base currently contains two facts: $arp(ip10\_1\_1\_6, ip10\_1\_1\_1, 1, mac00\_00\_00\_00\_00\_00)$ and $tcp(ip10\_1\_1\_6, ip10\_1\_1\_1, 3952, 80, yes, syn, 64)$. By asking DLV to compute all possible answer sets, we end up with only two. Each answer set contains one possible operating system for 10.1.1.6, and they are Linux Red Hat 7.0 and Linux Red Hat 8.0. ♢

At the end of Example 5.6, one could wonder why DLV did not output an answer set where 10.1.1.6 is assigned both Windows XP and SunOS. This would indeed be an answer set, but the weak constraint discussed at the beginning of Section 5.3.3.1 prevents a machine from being assigned multiple OSes when one is sufficient. However, there are some situations where the knowledge base will contain facts for which it is not possible to assign a single OS to a given machine and still form a consistent answer set, for instance if a group of computers is behind a NAT (network address translation tool) module. The weak constraint is flexible enough to handle such situations by differentiating between OS ambiguity and multiple OSes hidden behind a single IP address.

5.3.3.2.1 Knowledge Base Design We can represent the knowledge in two different ways: using a single knowledge base or using one knowledge base per computer. The former has the advantage of being very simple while the later is more efficient. When using a single knowledge base (see Figure 5.2(a)), every packet is recorded in a single EDB file. However, the number of answer sets will suffer from combinatorial

explosion. Using multiple knowledge bases (see Figure 5.2(b)) allows us to focus on a single computer and limits the number of answer sets, see Example 5.7.



(a) Single KB (b) Multiple KB

Figure 5.2: Knowledge Base Options

Example 5.7 (answer set combinatorial explosion)
Consider two computer A and B and a set of packets such that the set of possible OSes for A has size 11 and the one for B has size 15. If we use a single knowledge base containing every packet, then we need $165 = 15 \times 11$ answer sets to represent all the possibilities. On the other hand, if we use one knowledge base per computer, then we need only $26 = 11 + 15$ answer sets to represent every possibilities. Note that in both cases we end up with the same knowledge. ♢

To have a better idea of the impact of choosing between single or multiple knowledge bases, we measured the number of answer sets and the time required to compute them for some examples. For this experiment we used a set of packets generated by 7 hosts and we measured two values: the total number of answer sets and the time required to compute them. We computed those metrics for 7, 14, 21, 42, 84, and 140 packets[5]. For the single knowledge base case, we simply executed *DLV* to compute the answer sets and timed the execution. For the multiple knowledge base case, we called the *DLV* resolution engine for each of the 7 EDB files and added the time and number of answer sets of each execution to get the total cost of asking for an overview

[5]The number of packets is equally distributed among the 7 hosts, so 7 packets means 1 packet per host

Mode	Metric	Nb Packets					
		7	**14**	**21**	**42**	**84**	**140**
Single KB	**time (ms)**	5000	2563	203	78	94	94
	answer sets	6000	4608	240	8	2	2
Multiple KB	**time (ms)**	359	344	360	344	328	313
	answer sets	38	24	17	10	8	8

Table 5.3: Time Benchmarks

of the whole knowledge. The tests were fun on a Pentium 4 1.9 GHz computer running *Windows* 2000 sp4 with 784 Mo of RAM. The results for this experiment are shown in Table 5.3.

First, we can compare the number of answer sets produced by the two modes. In both cases, the number of answer sets decreases as the number of packets increases. This is due to the fact that with just a single packet it is hard to have a precise idea of the actual operating system on a machine: the less information we have, the more possible operating systems we get, and the more answer sets we need to describe the knowledge base. By comparing the number of answer sets for the two modes, we see two things:

- the maximum number of answer sets is much smaller in the multiple KB case, never more than 40 compared to a worst case of 6,000;
- the minimum number of answer sets is a little smaller in the single KB case (2 vs 8).

The first observation is explained by the combinatorial explosion experienced with a single knowledge base, see Example 5.7. The second observation, while surprising at first, can be explained by recalling that in the single KB case we compute the answer sets for one EDB file (in the best case this could give one answer set); while in the multiple KB case we compute the answer sets for seven EDB files (so the best case is seven answer sets).

Second, we can compare the computation time required by the two modes. With a single knowledge base, the time dramatically decreases (from more than 5 seconds to less than 0.1 second) as the number of packets increases. This can be explained by

the number of answer sets required when we only have a few packets. With multiple knowledge bases, the time is much more stable (always between 0.3 and 0.4 sec). Recall that the number of answer sets produced was also much more stable in the distributed mode (between 8 and 38). It is interesting to see the small amount of time required when using multiple KBs (0.4 seconds is always sufficient[6]) and that the number of packets seems to have little influence for the multiple KB case.

To avoid a combinatorial explosion of answer sets, we maintain one EDB file for each monitored IP address (and so we have multiple knowledge bases). Thus each packet can be placed in two different files (the file for the sender and the one for the receiver).

5.3.3.3 Queries for the Passive Module

While other passive fingerprinting tools are designed to guess an OS for each packet, leading to a diverging set of possible OSes, HOSD provides a converging set of possible OSes by eliminating candidates. To see the benefits of having a convergent set of possible OSes, we show how two different systems would answer three different queries (each time, P represents the set of possible OSes given by a tool).

"Is machine I running the OS o?"

- P **converges:** if $P = \{o\}$, then the answer is *yes*; if $o \notin P$, then the answer is *no*; otherwise the answer is *unknown*.
- P **diverges:** the answer is always *unknown* (a new packet may always come in to add a guess, which could be o or any other OS).

"Is machine I running an OS $\in O$?"

- P **converges:** if $P \subseteq O$, then the answer is *yes*; if $P \cap O = \emptyset$, then the answer is *no*; otherwise the answer is *unknown*.
- P **diverges:** the answer is always *unknown*.

[6]Remember, this is the time required to do 7 computations. So each individual computation takes about 0.06 second.

"Which OS is running on I?"

- P **converges:** if $P = \{o\}$, then the answer is o. Otherwise, the answer is *unknown*.
- P **diverges:** the answer is always *unknown*.

Of course, it is possible to approximate the answers when P does not converge. For instance, one could assume that o is the actual OS whenever $P = \{o\}$. However, this is not safe reasoning, as a new packet can change this answer. The divergent behavior of classic passive OSD tools is very problematic for the kind of automated reasoning required by IDS context gathering. However, the tools were designed to be used by human network administrators who are more flexible in the interpretation of the results.

5.3.4 Implementing the Active Module using ASP

When a query is made to the passive module (see Section 5.3.3.3) and the answer turns out to be *unknown*, then the task of the active module is to come up with a small set of active OS discovery tests, such that after their execution, the knowledge base will have enough information to answer the query properly. Here, we explain how ASP can be used to compute the *possible outcomes* of a test in a given knowledge state. A possible outcome for test T is a set of OSes that we could consider (depending on the outcome of T) as the set of possible OSes after the execution of T. This information can then be used to select the best test (or the best sequence of tests) to execute next for a given query. Note that the active module of HOSD is not implemented yet (this will be discussed in Section 7.9).

Section 5.3.4.1 presents a possible IDB file containing the ASP description of some actions (the active OS discovery tests). Section 5.3.4.2 describes how the system could be queried for test selection and provides an example of an EDB file containing the description of the initial situation. Finally, Section 5.3.4.3 discusses the kinds of queries that will be supported by the active module vs other active tools.

5.3.4.1 Active IDB

A fragment of a possible intentional database (*activeOSdiscovery.IDB*) for the active module is presented in Figure 5.3. The first four rules[7] define the test outcome interpretation module; their meaning is as follows:

1) if an association of an IP address to an OS $\langle I, OS\rangle$ does not hold before the execution of a test, it will not hold after its execution;

2) if an association $\langle I, OS\rangle$ holds before the execution of a test and the outcome of this test confirms that possibility, $\langle I, OS\rangle$ will still hold after the execution of the test;

3) all associations $\langle I, OS\rangle$ that are true remain true when no test is executed;

4) what is not explicitly said to be possible by the outcome of a test should be considered not possible.

The second part of Figure 5.3 describes the possible outcomes of the active test $T8$. Note that $T8$ corresponds to the passive knowledge updating rules presented in the **TCP SynAck** group of Figure 5.1. If $T8$ is executed against I at time T, this will cause at least one (and exactly one) of the predicates $oT8A(I, \tau_1)$, $oT8B(I, \tau_1)$, $oT8C(I, \tau_1)$, or $oT8D(I, \tau_1)$ to be true ($oT8A$ denotes outcome A of test $T8$). Each outcome then provides the OSes that are possible after the execution of $T8$. For instance, MacOS and SunOS are the possible explanations of outcome C for $T8$. In other words, if the execution of $T8$ leads to outcome C, then the target could not be running an OS other than MacOS or SunOS.

The set of possible OSes for an outcome of a test is the intersection of what was possible before the execution of the test with what is possible with respect to that specific outcome (see rule 2 in Figure 5.3). The actual outcome of a test depends on the test result and thus on the actual operating system running on the machine. Example 5.8 details the effects of an action. A complete example of test selection will be presented in Section 5.3.4.2.

[7]The predicate *holds/3* expresses the possible OSes at each state, while *possible/3* denotes what is possible with respect to the outcome of a test.

%Test Outcome Interpretation
1) -holds(I,OS,τ_1) :- -holds(I,OS,τ), $\tau < \tau_1$.
2) holds(I,OS,τ_1) :- holds(I,OS,τ), next(τ,τ_1), possible(I,OS,τ_1).
3) holds(I,OS,τ_1) :- holds(I,OS,τ), next(τ,τ_1), not actionExecuted(I,τ).
4) -possible(I,OS,τ) :- not possible(I,OS,τ).

%T8 - The TCP SynAck test
oT8A(I,τ_1) v oT8B(I,τ_1) v oT8C(I,τ_1) v oT8D(I,τ_1) :-
next(τ,τ_1), execute(testT8,I,τ).

%oT8A (OS is linux Red Hat5.2)
possible(I,linuxRH5_2,τ_1) :- oT8A(I,τ_1).
%oT8B (OS is windows 2000 or XP)
possible(I,win2k,τ_1) :- oT8B(I,τ_1).
possible(I,winXP,τ_1) :- oT8B(I,τ_1).
%oT8C (OS is mac or sun)
possible(I,macOS,τ_1) :- oT8C(I,τ_1).
possible(I,sunOS,τ_1) :- oT8C(I,τ_1).
%oT8D (OS is freeBSD 5.0, linux RedHat 7.1 or 8.0)
possible(I,freeBSD5_0,τ_1) :- oT8D(I,τ_1).
possible(I,linuxRH7_1,τ_1) :- oT8D(I,τ_1).
possible(I,linuxRH8_0,τ_1) :- oT8D(I,τ_1).

Figure 5.3: Some Rules for Active OS Discovery

Example 5.8 (effects of actions)
Suppose that $T8$ is executed against I at time τ, where we know that I is running Linux Red Hat, but we do not know if it is running 5.2, 7.1, or 8.0. Then at time τ_1, exactly one of $oT8A(I,\tau_1)$, $oT8B(I,\tau_1)$, $oT8C(I,\tau_1)$, or $oT8D(I,\tau_1)$ will be true. Now assume I is actually running Linux Red Hat 7.1; the outcome of the $T8$ is then $oT8D(I,\tau_1)$. With the definition of **oT8D** from Figure 5.3, we can see that $possible(I, freeBSD5\_0, \tau_1)$, $possible(I, linuxRH7\_1, \tau_1)$ and finally $possible(I, linuxRH8\_0, \tau_1)$ are forced to be true. Thus, we can conclude that I is running Linux Red Hat 7.1 or 8.0. I cannot be running Linux Red Hat 5.2, as the test result eliminates it (*possible/3* is not true in rule 2) and cannot be running Free BSD 5.0 as it was not a possibility before the execution of the test (*holds/3* is not true in rule 2). ♢

```
holds(ip10_1_1_6, macOS,0).
holds(ip10_1_1_6, sunOS,0).
holds(ip10_1_1_6, linuxRH5_2,0).
holds(ip10_1_1_6, linuxRH7_1,0).
execute(testT8,ip10_1_1_6,0).
```

Figure 5.4: Computing Test Outcomes

5.3.4.2 Active EDB and Querying

Here we explain how our knowledge base can be queried using ASP to provide the possible outcomes of a test.

We need to specify two components for the extensional database (*activeOSDiscovery.EDB*): the current knowledge situation (i.e., the current set of possible OSes) and the goal (i.e., the test for which we want the possible outcomes). Then, we can use DLV to compute the answer sets of the program formed by the IDB file presented above and the EDB file discussed here. Each answer set will correspond to a possible outcome for the given test. Example 5.9 provides more information regarding the querying mechanism and the content of the EDB file.

Example 5.9 (active module with ASP)
Assume we are in a situation where the set of possible OSes for a specific computer (10.1.1.6) is { Linux Red Hat 7.1, MacOS, SunOS, Linux Red Hat 5.2 }. Let's consider the execution of Test T8 as presented in Figure 5.3 and see how we can compute the outcomes of that test using ASP. Figure 5.4 shows an EDB file containing the required information. The first part contains facts about the current set of possible OSes (e.g., at time 0 MacOS is possible), while the second one specifies which test we are interested in (here we execute T8 on 10.1.1.6 at time 0). Now if we compute the answer sets using DLV, we obtain: $\{MacOS, SunOS\}$, $\{LinuxRedHat5.2\}$, and $\{LinuxRedHat7.1\}$; these corresponds to the possible outcomes of test T8. $\diamondsuit$

Once the possible outcomes are computed for every test, it is possible to select the *best* test to be executed next by measuring how *close* to the goal each test will bring us. Of course, we can use several measures for *best* and *close*, which lead to different test selection strategies. This will be discussed in Chapter 7.

Note that the ideas presented in Example 5.9 can easily be extended to compute the possible outcomes of a series of tests.

5.3.4.3 Queries for the Active Module

State-of-the-art active tools for OS discovery have the single goal of learning which operating system is running on a given machine. We believe this to be a limitation, one that will not be present in HOSD due to its knowledge-based design. The intuition is that it is usually easier (i.e., it requires fewer tests) to answer the query "Is machine I running the OS o?" or "Is machine I running an OS $\in O$?" than to learn the exact OS running on I; see [15].

Different queries can be implemented using different test selection strategies. For instance, the distance from a set of possible OSes to the goal will be measured differently for each query as their goals differ.

5.4 Experimental Results

In this section we discuss the results of three experiments run with the passive module of our tool, POSD for Passive OSD. We start, in Section 5.4.1, with the IDS context gathering experiment presented in Chapter 2 to compare POSD with existing tools. Then, in Section 5.4.2, we consider the experiment of Chapter 3 which captures more naturally the objective of OSD tools, i.e., to obtain the actual OS running on the computers. Finally, in Section 5.4.3, we provide time benchmarks to see if our ASP implementation is practical.

5.4.1 IDS Context Gathering

We reran the IDS context gathering experiment introduced in Chapter 2 with POSD. The objective is to compare the knowledge-oriented approach with other classical approaches to see if it is a promising avenue.

Figure 5.5 and Table 5.4 present a summary[8] of the results for POSD. It also includes, for comparison purposes, the results for the other classical tools and for the

[8]Detailed results are presented in Appendix A, Section A.4.

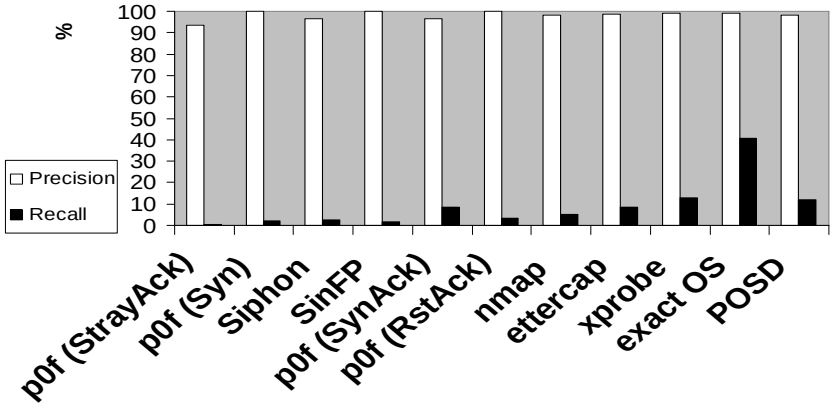


Figure 5.5: Precision/Recall Comparison for POSD

Table 5.4: Precision/Recall Summary Comparison for POSD

Measure / Tool	Precision	Recall
p0f (StrayAck)	29/31 (93.55%)	29/4575 (0.63%)
p0f (Syn)	1/1 (100.00%)	1/4575 (2.19%)
Siphon	109/113 (96.46%)	109/4575 (2.38%)
SinFP	75/75 (100.00%)	75/4575 (1.64%)
p0f (SynAck)	399/414 (96.38%)	399/4575 (8.72%)
p0f (RstAck)	156/156 (100.00%)	156/4575 (3.41%)
Nmap	232/236 (98.31%)	232/4575 (5.07%)
ettercap	387/392 (98.72%)	387/4575 (8.46%)
Xprobe	583/589 (98.98%)	583/4575 (12.74%)
exact OS (see Table 2.4)	1862/1875 (99.31%)	1862/4575 (40.70%)
POSD	559/568 (98.42%)	559/4575 (12.22%)

case where we know the actual OS.

5.4.1.1 Precision

From the precision point of view, POSD results are very similar to other tools. Once again, some mistakes are due to the erroneous entries in Security Focus. However, it seems that POSD has made a few other mistakes, these require further investigation.

5.4.1.2 Recall

From the recall point of view, POSD performs extremely well. POSD is significantly better than any other passive tool. Moreover, POSD is very close to the best active (and best overall) tool: Xprobe. POSD obtains a 12.22% recall while Xprobe has 12.74%. These great results for POSD can be explained mainly by two features:

F_1 POSD has a memory, allowing it to rely on knowledge acquired on previous events.

F_2 POSD is multi-packet based, which allows it to extract more information on stimuli-response events.

5.4.1.3 Discussion

The results above confirm that our knowledge-oriented approach is an improvement over classical tools. However, POSD is still far from achieving the potential of having OS information for IDS context (12% vs 41%). We believe that adding the active module on top of POSD will dramatically improve the overall results, beyond what any other classical tools can do. The following features (together with the two features mentioned above) of our hybrid approach support our intuition:

F_3 HOSD can take advantage of both the passively and actively gathered information. Some events, such as a SYN packet from the target, cannot be forced by a stimulus and thus cannot be used in active-only OSD. Thus this will be an improvement over current active tools. Obviously, some events will not be available passively, and so HOSD can trigger them actively.

F_4 Because active tools always execute all their tests before analyzing the results, they must limit their set of tests to a minimum (to avoid sending too many packets). HOSD, on the other hand, can enjoy a wide selection of tests[9] (as only a few of them are executed each time). This will allow HOSD to obtain the information it needs in most situations.

[9]HOSD can also rely on several instance of the same test, with different field values, to obtain more precise information.

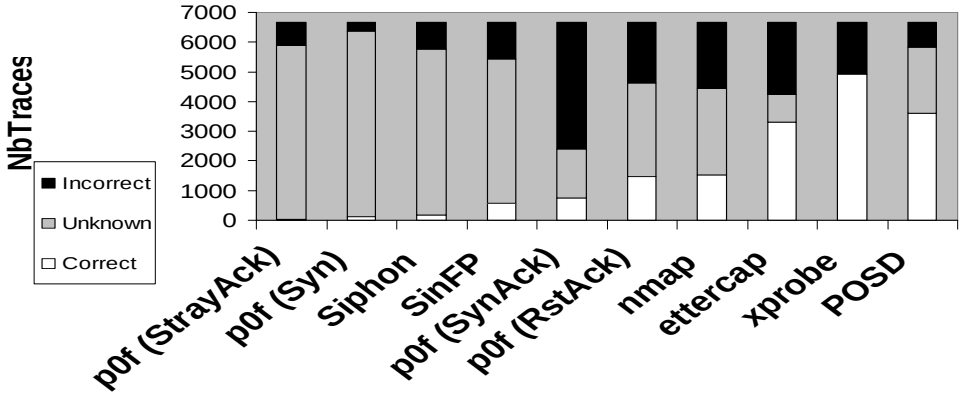


Figure 5.6: Recall Comparison for POSD

5.4.2 OSD Experiment

In Section 3.5 of Chapter 3 we presented another experiment to measure the accuracy of OSD tools. Here, we revisit this experiment, now including results for POSD.

5.4.2.1 Recall

Figure 5.6 provides an overview of the recall of POSD, including the other tools for comparison (the actual numbers can be found in Table 5.5). Let's recall that the white area (correct answer) represents the number of traces for which the set of possible OSes provided by a tool contains the actual OS; the grey area (unknown) represents the number of traces for which the tool does not take any guess (or does not eliminate any OS, in the case of POSD); and the black area (incorrect answer) represents the number of traces for which the set of possible OSes does not contain the actual OS.

POSD has the second best recall (see white area in Figure 5.6) behind Xprobe. It is thus better than all other passive tools. It is important to note that among the five tools having a recall over 20% (p0f in RstAck mode, Nmap, Ettercap, Xprobe, and POSD), POSD is the most *reliable* one (see black area in Figure 5.6). That is, POSD provides the incorrect answer for only 13% of the cases, while the closest tool provides the incorrect answer over 25% of the time. This is not surprising since POSD is designed to eliminate an OS only when it cannot be running on the target computer.

Table 5.5: Recall Comparison for POSD

Tool \ Measure	Correct Answers	Unknown Answers	Incorrect Answers
p0f StrayAck	28 (0.42%)	5847 (87.85%)	781 (11.73%)
p0f Syn	135 (2.03%)	6244 (93.81%)	277 (4.16%)
Siphon	183 (2.75%)	5574 (83.74%)	899 (13.51%)
SinFP	562 (8.44%)	4878 (73.29%)	1216 (18.27%)
p0f SynAck	746 (11.21%)	1661 (24.95%)	4249 (63.84%)
p0f RstAck	1461 (21.95%)	3159 (47.46%)	2036 (30.59%)
Nmap	1534 (23.05%)	2922 (43.90%)	2200 (33.05%)
ettercap	3294 (49.49%)	955 (14.35%)	2407 (36.16%)
Xprobe	4933 (74.11%)	0 (0.00%)	1723 (25.89%)
POSD	3615 (54.32%)	2206 (33.15%)	834 (12.54%)

Table 5.6: Precision Summary Comparison for POSD

Tool \ Measure	Average Size of Set of Possible OSes
ettercap	21.37
p0f RstAck	18.58
Xprobe	12.30
Nmap	4.86
POSD	16.95

5.4.2.2 Precision

To make sure the recall results presented above are legitimate, we must also consider the precision results of POSD. Table 5.6 provides the average size of the set of possible OSes given by POSD[10] (and the tools with recall over 20%, for comparison) when providing the correct answer. We can see that POSD has a better precision than the two passive tools considered here. Both active tools (Nmap and Xprobe) provide a better precision than POSD. This is not unexpected, since active tools can fetch more precise information than passive tools.

[10] Actual sizes for POSD can be found in Section C.2 of Appendix C

5.4.2.3 Discussion

Once again, the results confirm that our knowledge-oriented approach is an improvement over classical passive tools. We believe that adding the active module on top of POSD will dramatically improve the precision of POSD (since better information will be available). It should also improve recall since information can be fetched even when the trace does not contain any relevant information (most of the grey area of Figure 5.6 should turn white). We will also investigate the cases where POSD made a mistake (the black area of Figure 5.6) to see where the error occurred (it is probably due to erroneous fingerprints).

5.4.3 Time Benchmarks

One of the main concerns with our current implementation is the time complexity. Obviously, the expressiveness of our knowledge representation language (ASP) has a computational cost. In [18], we discussed a few optimization techniques:

- Keeping one set of packets for each IP address (i.e., one EDB file per IP address) to minimize the combinatorial explosion; and

- Preventing the set of facts from growing too much by computing the set of possible OSes for every X packets and then discarding those packets, keeping only a summary[11] of the current set of possible OSes.

In this section we present the time required by the passive module to compute the set of possible OSes as well as the time required by the active module to obtain the answer to the query (including test selection, test execution, and test result analysis). The experiment was run on 6,656 traffic traces[12] using an Intel Core 2 6300 CPU @ 1.86 Ghz with 2 Gb of RAM running Windows 2000 and relying on DLV [23] as the engine for evaluating ASP programs. A coarse summary of time results are shown in Table 5.7.

[11] The summary is a formula of the form $\bigvee_{o \in P} os(I, o)$, where P is the set of possible OSes and I is the IP address currently monitored.

[12] We used the traces for the OSD experiment discussed in Section 5.4.2.

Table 5.7: Time Benchmarks (in ms)

Time / Tool	Min	Average	Max
POSD	102	386	16,123

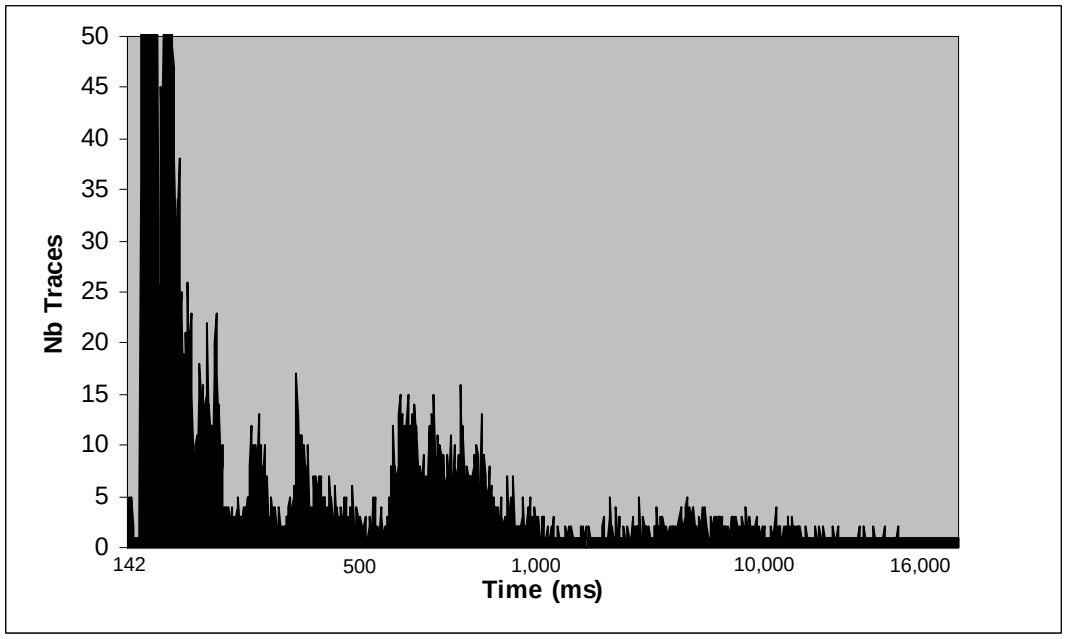


Figure 5.7: Time Distribution

Although the minimal and average computation time (respectively 102 and 386 milliseconds) required are quite acceptable, the worst-case time (16 seconds) is definitely not.

Figure 5.7 presents the computation time distribution across the 6,656 traces. Moreover, Table 5.8 provides a summary of that distribution. We can consider that any computation time above 1 second is inadequate (and we have 463 such cases).

Because the passive module will be a continuously running background task (possibly on a dedicated computer), the main constraint on the passive module is to keep up with the relevant portion of the network traffic. We consider only the packet types that are interesting for the deduction rules. For instance, we avoid packets generated by file transfer, gaming, and streaming. Moreover, we can sample randomly when under heavy traffic (unlike intrusion detection systems, an OS discovery tool can

Table 5.8: Time Distribution summary

Time Range (ms)	Nb Traces
0-499	4948
500-999	1245
1,000-9,999	433
10,000+	30

drop traffic without consequences). Nevertheless, the worst-case time is such that the current implementation is not suitable for real-world application.

5.5 Conclusion

In this chapter, we proposed a new hybrid approach to OS discovery. Our approach addresses several drawbacks of the classical approaches and tools, while keeping their advantages.

We provided a proof-of-concept implementation of the passive module (POSD) and compared it with several existing OSD tools. In both experiments, POSD outperformed every other tools but one, Xprobe. These results support our intuition that OSD tools would benefit form a knowledge-oriented approach.

We expect our hybrid tool to be much better than any other OSD tool once it is complete, i.e., when the active module is built on top of the passive one.

5.6 Contributions

This chapter provides two major contributions:

- The elaboration of a better approach to operating discovery.
- An open source tool, HOSD, available in pre-alpha release from `http://hosd.sourceforge.net`.

Accepted and/or pending publications with respect to this chapter are:

- [18]: 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.

- [13]: François Gagnon. Operating System Discovery Using Answer Set Programming. Advanced Course on Artificial Intelligence Summer School - Poster Session (ACAI'07), 2007.

- [17]: François Gagnon and Babak Esfandiari. Gathering Context for Intrusion Detection - A Study of Operating System Discovery. Submitted for Journal Publication to IEEE Transactions on Network and Service Management.

5.7 Proposal

Concerning the topic presented in this chapter, we propose to work on the following for inclusion in the final thesis:

- faster implementation of the passive module
- implementation of the active module
- experiment to compare our full hybrid tool with other OSD tools
- design a new experiment to evaluate OSD tools in the context of the Single OS Query, i.e., "Is the computer running the specific OS *o*?".

Since the implementation of the passive approach suffers from time complexity, we will provide a more effective implementation based on an algorithm borrowed from diagnosis theory and discussed in Chapter 6.

Currently, only the passive module is implemented. We will provide an implementation of the active module[13] (integrated with the passive module). For the active module, we will need at least one algorithm for each query. This implementation will also be based on algorithms from the diagnosis theory, these algorithms are discussed in Chapter 7.

The implementations of both the passive and active modules will be integrated to provide a full hybrid OSD tool. We will compare that tool with the existing OSD

[13]This requires, among other things, a packet injection utility.

tools based on our OSD experiments. We expect our hybrid tool to be much better than any other OSD tools.

So far, we have provided two experiments. One measures the ability of a tool to solve the Group OS Query ("Is the computer running an OS in the given set O"). The other measures the ability of a tool to solver the Exact OS Query ("Which OS is running on the computer?"). We will design another experiment to evaluate the ability of a tool to solve the Single OS Query ("Is the computer running the specific OS o").

Chapter 6

Diagnosis - Candidate Generation for Passive Operating System Discovery

Essentially, all models are wrong, but some are useful.
-George Box

Abstract

In the previous chapter, we proposed a new approach to operating system discovery. We also provided a partial implementation of this approach using answer set programming. However, our implementation suffers from the time complexity of ASP, which can solve Σ_2^P-complete problems[1] [36]. We do not know yet what is the complexity of the OS discovery problem, but we expect it to be lower than that. In this chapter, we model OS discovery as a diagnosis task and derive a fast algorithm for the passive module of HOSD. Diagnosis problem solving will also provide insights for the implementation of the active module.

6.1 Introduction

The task of operating system discovery is to find, for a given computer, an operating system that explains the observed behavior of that computer (i.e., the packets sent during communication). This task is exactly what is studied by the theory of diagnosis problem solving. In this chapter, we show how OS discovery can be modeled as a diagnosis task. This formalization of OS discovery will serve as our guideline in the final implementation of our HOSD tool. We want to provide a strong theoretical background for HOSD for the following reasons:

- It will give us insights regarding the complexity of our problem.
- It will provide us with tested algorithms to implement the different modules.

[1] Σ_0^P = P and has no source of intractability, Σ_1^P = NP and has one source of intractability, $\Sigma_2^P = NP^{NP}$ (i.e., the class of problems that are solvable in polynomial time on a non-deterministic Turing machine provided we have a constant time oracle for the problems in NP) and has two sources of intractability; see [51].

- Improvements and/or extensions in the general theory will automatically enhance the tool.
- It will serve as a formal specification of HOSD's engine.

The rest of the chapter is structured as follows: Section 6.2 introduces the theory of diagnosis problem solving. Then, Section 6.3 explains how operating system discovery can be seen as a diagnosis task. Section 6.4 adapts the main diagnosis algorithm (candidate generation) to the specific task of OS discovery. This leads to a polynomial algorithm for the active module of HOSD. The chapter will end with a short conclusion, a summary of the contributions made throughout this chapter, and the proposed work, related to the topic of this chapter, to be done during the thesis.

6.2 Diagnosis Background

Intuitively, the goal of a diagnosis engine is to find the broken components of a system, if any. We can easily generalize that goal to finding the explanations as to why a system behaves in a given way. This second view of diagnosis is much closer to our OSD problem.

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.

6.2.1 Diagnosis Problem Specification

Based on [60], 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, these could 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 could be disease symptoms.

SD: The system description provides the information for diagnosing the system, i.e., it provides a way, direct or indirect, 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 6.1 illustrates the general behavior of a diagnosis tool. 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 allergy.

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

D: If yes, then the work is completed.

E: Otherwise, like 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 determining the actual diagnosis. For instance, the doctor could check the glands of the patient. The selected test is executed. This will stimulate the system and generate new observations and we go back to step A.

Notice the close relationship between the operation of a diagnosis tool and the hybrid approach to OS discovery presented in Chapter 5. The passive module of HOSD corresponds to steps A and B discussed above, while the active module matches steps E and F. There is, however, one thing which does not match our requirements for HOSD: the user cannot interact with the diagnosis tool to ask a specific query. Chapter 7 will extend the simple diagnosis model presented above to address this limitation.

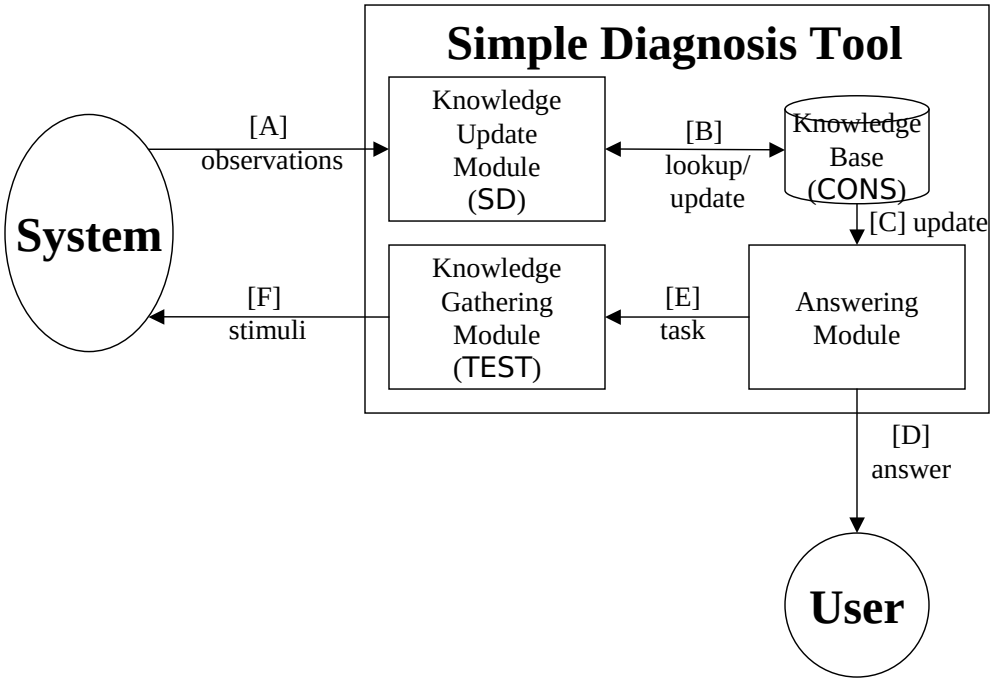


Figure 6.1: Simple Diagnosis Tool Behavior

6.2.2 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 6.1 (Hypothesis Space)
The hypothesis space, 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 $\mathcal{H} = \wp(\mathsf{CONST})$ or $\mathcal{H} = \{\{c\}|c \in \mathsf{CONST}\} \cup \{\emptyset\}$ (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 several explanatory constituents. In that case, $\mathcal{H} = \wp(\mathsf{CONST})$. In both cases, $\mathcal{H}$

contains sets of explanatory constituents.

Definition 6.2 (Diagnosis Candidate)
A diagnosis candidate (or candidate for short) is a hypothesis from $\mathcal{H}$ explaining the given observations. For a given set of observations, there might be several candidates. Each candidate is a subset of CONST, and we use $\Gamma(\Theta)$ to denote the set of diagnosis candidates of observations Θ. ◯

Definition 6.3 (Actual Diagnosis)
The actual diagnosis 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. This corresponds to steps A and B in Figure 6.1. Candidate elimination, on the other hand, 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). This corresponds to steps E-F in Figure 6.1. In this chapter, we focus on candidate generation. The discussion on candidate elimination is delayed until Chapter 7.

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

Given a diagnosis problem $\langle\ \mathsf{CONST}, \mathsf{SD}, \mathsf{OBS}\ \rangle$, an instance of that problem is a set of observations $\Theta \subseteq \mathsf{OBS}$.

Definition 6.4 (Consistency-Based Explanation)
Given a diagnosis problem $\langle\ \mathit{CONST}, \mathit{SD}, \mathit{OBS}\ \rangle$, a consistency-based diagnosis candidate for some given observations $\Theta \subseteq \mathit{OBS}$ is $\Delta \in \mathcal{H}$ such that:

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

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 the constituents of

Δ to be have property P, while the others don't, is consistent with the observations. The consistency-based concept of an explanation was introduced in [60]. ◯

Definition 6.5 (Abductive Explanation)
Given a diagnosis problem $\langle$ CONST, SD, OBS $\rangle$, an abductive diagnosis candidate for some given observations $\Theta \subseteq$ OBS is $\Delta \in \mathcal{H}$ such that:

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

and

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

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[2] in [56]. ◯

Abductive 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 [60], 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 [59], SD contains rules directly associating the observations and the explanatory constituents (this is considered as low-level information). Note that model and rule-based diagnosis can use both abductive and consistency-based reasoning. This produces four families of diagnosis problems. Section 6.2.3 discusses these families in more details.

6.2.3 Four Diagnosis Families

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

[2] [31] provides a slightly different definition of abductive explanation, omitting the consistency check. However, they are not always equivalent and the definition presented here is preferable.

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 [52]). Thus is it 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. It is difficult to provide a general definition of the two kinds of knowledge; for model-based diagnosis, we talk about normal vs abnormal behavior knowledge (see [56]), while for rule-based diagnosis we talk about explanatory vs fault-descriptive knowledge (see Example 6.1 below). Normal behavior knowledge and explanatory knowledge require consistency-based reasoning, while abnormal behavior knowledge and fault-descriptive knowledge require abductive reasoning. Example 6.1 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 [56, 57, 58].

Example 6.1 (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. 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. ♢

Below, we study some of these families in more details. In Section 6.2.4 we present Reiter's approach to diagnosis. This approach is model-oriented and relies on consistency-based reasoning. Among other things, we study the candidate elimination algorithms proposed by Reiter; we will use this as a base when designing our candidate elimination algorithm specific to OS discovery in Section 6.4. 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. In Section 6.2.5 we study the work by Poole regarding the different reasoning mechanisms (abductive and consistency-based) to chose the appropriate one for OSD.

6.2.4 Reiter's Model-Based Diagnosis

Here we explain in more detail the classic approach to diagnosis proposed by Reiter in his seminal paper [60]. Reiter's approach is the most well-known in the diagnosis community. It will be the base for our new algorithm for the passive module of HOSD.

Reiter's approach to diagnosis is model-based and mainly suitable for engineering (e.g., circuits diagnosis). The knowledge in SD is of the normal behavior kind, 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 6.2.

Example 6.2 (full adder (taken from [60]))
The device, shown in Figure 6.2 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. ♢

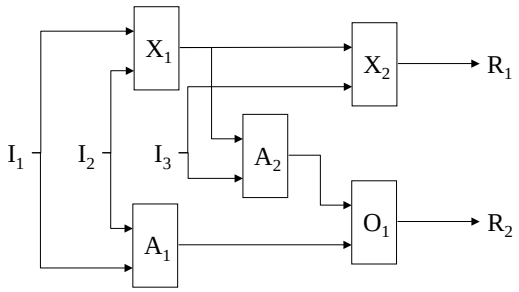


Figure 6.2: Full Adder Device

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

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

OBS: $\langle I_!, I_2, I_3, R_1, R_2\rangle$ where each I_i and R_i has a binary value (0 or 1), see Example 6.3.

SD: As presented in Table 6.1 the system description contains:

- 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 6.3 (full adder (continued from Example 6.2))
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:

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

$\diamondsuit$

Table 6.1: SD for the Full Adder

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

Based on the consistency-based concept presented in Definition 6.4, 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 6.4 (full adder (continued from Example 6.3))
The following observations Θ for the binary full adder conflicts with the expected behavior of the system.

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

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[3]. $\Delta = \{X_1\}$ is a diagnosis candidate for this instance, as assuming that X_1 is abnormal while every other component is normal makes the theory consistent. 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

[3]More formally, the system is faulty because $\Delta = \emptyset$ is not a diagnosis candidate for the given observation.

X_2, we cannot predict both $out(X_2)$ and $out(A_2)$. 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 6.2. $\diamondsuit$

Table 6.2: Diagnosis Candidates for Example 6.4

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

Example 6.4 illustrates that a simple diagnosis problem can have several candidates. Moreover, Table 6.2 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. This property is proven in [60], see Proposition 3.4. In the worst case, there could be exponentially many candidates. To partially circumvent this problem, Reiter considers only set-minimal candidates. Table 6.3 shows all set minimal candidates for Example 6.4

Table 6.3: Minimal Diagnosis Candidates for Example 6.4

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

6.2.4.1 Computing Diagnosis Candidates

The definition of a diagnosis candidate, Definition 6.4, 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 6.3, 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.

Figure 6.3: Naive Algorithm to Computer Diagnosis Candidates in Multiple Faults

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

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 [60], 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 new algorithm for the passive module of HOSD.

Definition 6.6 (Conflict Set)
A conflict set for a diagnosis problem instance is a set $\{c_1, \ldots, c_k\} \subseteq \mathsf{CONST}$ *such that*

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

is inconsistent. A conflict set is minimal iff it has no proper subset that is also a conflict set. ◯

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

Definition 6.7 (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 6.1 (Theorem 4.4 in [60])
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 [60]. □

Proposition 6.1 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).

The algorithm presented here contains three sources of intractability:

- the exponential size of $\mathcal{H}$.
- the consistency-check procedure.
- the hitting set procedure.

In [60] Section 4, Reiter proposes an algorithm to compute the minimal diagnosis based on a pruned hitting set data structure. However, it is not clear that this new algorithm has a better worst-case scenario than the algorithms described above.

6.2.5 Rule-Based Diagnosis

As discussed previously, OS discovery cannot be easily represented using model-based diagnosis. For that reason, 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 and fault-descriptive) and the format of the rules used to encode that knowledge. Following [56], we start by providing the format of the rules used to encode each kind of knowledge in SD (explanatory and fault-descriptive).

Definition 6.8 (Rules for Explanatory Knowledge)
Each rule gives the possible causes (explanations) for a specific observation. Rules have the form:

$$EX(c_1) \vee EX(c_2) \vee \ldots \vee EX(c_n) \leftarrow \theta_1 \tag{6.1}$$

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

Definition 6.9 (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:

$$\theta_1 \leftarrow EX(c_1) \tag{6.2}$$

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

$$\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n \leftarrow EX(c_1) \tag{6.3}$$

◯

Now we will compare the two kinds of knowledge and select the one appropriate to OS discovery.

6.2.5.1 Intuitive Meaning of the Rules

Rule 6.1 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 6.3 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. 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". 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.

6.2.5.2 Handling Incomplete Knowledge

We consider three situations where we can have incomplete knowledge:

Unanticipated Explanatory Constituent: In this case, 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: Second, we can have the explanatory constituent c and the observation θ in the model, but fail to 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: Finally, OBS does not include every observation we can make on the system. For instance, the fact that we can observe the sex of the patient is not incorporated in 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.

6.2.5.2.1 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[4]. In general, we will end up with diagnosis candidates of higher cardinality to compensate (by blaming additional constituents) for the observations caused by c.

6.2.5.2.2 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 constituents to compensate for the role of c).

6.2.5.2.3 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 6.4.

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 6.5), 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

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

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 will adopt the explanatory knowledge approach using consistency-based reasoning.

6.2.6 Diagnosis properties

Before describing how operating system discovery can be seen as a diagnosis task, we introduce properties that help defining a specific diagnosis task (or choose the proper model to represent it). Some of these properties can already be handled by the diagnosis framework presented in section 6.2.1, 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, see [60]. In the single fault case, at most one explanatory constituent can be used to explain the observations. In the multiple faults case, several constituents can be responsible simultaneously. Fault cardinality can be handled by most diagnosis frameworks.

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 6.2.3 and [21] proposes a general framework to handle both reasoning mechanisms simultaneously.

Stability The system can be *static* or *dynamic*. A static system does not change by itself over time, while a dynamic one might[5]. A dynamic system can develop

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

new problems (e.g., a patient 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 [7]). 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 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, you 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, than 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 [34] and [71]. 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 most systems, even before having any observation, some explanations are more likely than others (e.g., some diseases are more common than others). Otherwise, the prior probabilities of the explanation are all equal. Considerable work has been done on the use of prior probabilities of failure, mainly by Johan de Kleer in [24, 25, 27].

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.

Using the diagnosis problem representation discussed earlier in this chapter and the above properties, we can now characterize operating system discovery as a diagnosis task.

6.3 Operating System Discovery as a Diagnosis Task

In Chapters 3 and 5 we introduced different approaches (active, passive, and hybrid) to OS discovery. Here, we explain how the passive module of the hybrid approach can be modeled as a diagnosis task. As a result, we will be able to use the complexity results and the algorithms provided by the theory of diagnosis. Moreover, this will provide an elegant model to describe our hybrid approach, mainly the knowledge representation part and the important converging property of the set of possible OSes.

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

6.3.1 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* 2000 Sp1). 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 several computers are hidden behind a network address translator (NAT). In such a case, the traffic coming from those machines will appear to come from the NAT, but it will represent different OS behavior (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. Nevertheless, we decided to limit ourselves to a single fault model for the sake of simplicity[6].

6.3.2 Observations (OBS)

Observations in OSD are network events. For simplicity's sake, we consider here that an observation is a network 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.

6.3.3 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 terms of interactive constituents. One of the main critics against 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 4, 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 Chapter 8.

The rules composing SD will have the form

$$c_1 \vee c_2 \vee \ldots \vee c_n \leftarrow \theta \tag{6.4}$$

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 6.2.5. We could have used the rule format advocated by [56] ($\theta \leftarrow c$), encoding fault-descriptive

[6] We do not expect our OSD tool to encounter many NATs, especially when gathering IDS context, see discussion in Section 3.7.

knowledge; however, we believe rules like (6.4) to be better suited for our task. We based our decision on the discussion we had of Section 6.2.5 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 6.2.4.1, we can derive a fast algorithm for candidate generation in OSD. This algorithm will be presented in Section 6.4.

Based on the content of SD and the discussion in [58], we use the consistency-based definition of a diagnosis candidate for OSD.

6.3.4 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 6.2.6 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 take advantage of a multiple faults setting.

Fault Behavior Faults are clearly 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.

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 OS 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 Sun headquarters.

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 6.4 summarizes the diagnosis properties of OSD and emphasizes the assumptions we are making when modeling OSD for our implementation in HOSD.

Table 6.4: Diagnosis Properties for OS Discovery

	Actual OS Discovery	**Our Model**
Fault Cardinality	Multiple Faults	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

6.4 Candidate Generation Algorithm for OSD

Based on the candidate generation algorithm provided by Reiter in [60] (also discussed in Section 6.2.4.1) 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 6.4, 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 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 rule.

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

Figure 6.4: General Candidates Generation Algorithm

6.4.1 Algorithm Analysis Parameters

Before we provide the two algorithms and their analysis, let's 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, then |CONST| = 208. Once again, this number is mainly constant. It only changes when we integrate a new OS. Because they may have different vulnerabilities, we consider two versions of the same OS to be two distinct OSes, e.g., Windows 2000 sp1 and Windows 2000 sp2.

- |Θ| (the set of given observations). In our implementation, we will keep the size of Θ below a certain threshold, say 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).

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 $\mathcal{C} \leftarrow \emptyset$
2 FORALL $r \in \mathsf{SD}$
2.1 FORALL $\theta \in \Theta$
2.1.1 IF θ matches r
2.1.1.1 $\mathcal{C} \leftarrow \mathcal{C} \cup \{set_r\}$
3 RETURN $\mathcal{C}$

Figure 6.5: Conflict Sets Generation Algorithm

6.4.2 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.

Based on the above property, we present a polynomial time algorithm to compute the conflict sets, see Figure 6.5.

The algorithm of Figure 6.5 runs in $O(|\mathsf{SD}| \times |\Theta|)$, assuming the union operation performed in step 2.1.1.1 takes unit time[7].

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

6.4.3 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 6.1, 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 6.2
$\{h\}$ is a singleton hitting set of the collection $\mathcal{C}$ iff $h \in \cap_{S \in \mathcal{C}} S$.
Proof.

$\Rightarrow$: Assume $\{h\}$ is a singleton hitting set of the collection $\mathcal{C}$. By definition of a hitting set (see Definition 6.7), $h \in S$ 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 6.7), 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, see Figure 6.6.

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

[7]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 an indexed random access file.

HittingSetsGeneration($\mathcal{C}$)

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

```
1   H' ← ∅
2   H ← firstElem(C)
3   FORALL S ∈ C
3.1     H ← H ∩ S
4   FORALL h ∈ H
4.1     H' ← H' ∪ {{h}}
5   RETURN H'
```

Figure 6.6: Hitting Sets Generation Algorithm

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 run in $O(|\Theta| \times |\mathsf{CONST}|) + O(|\mathsf{CONST}|) = O(|\Theta| \times |\mathsf{CONST}|)$

The three sources of intractability discussed in Section 6.2.4.1 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).

6.4.4 Impact

The algorithm presented in Figure 6.4 runs in $O(|\mathsf{SD}| \times |\Theta|) + O(|\mathsf{CONST}| \times |\Theta|)$ which is polynomial. The impact of such a fast algorithm for candidate generation is direct to our OSD problem: we can use this polynomial algorithm for the passive

module of HOSD. This should provide a significant speed up compared to the current ASP implementation.

6.5 Conclusion

In this chapter, we formalized the OS discovery task as a diagnosis problem. This partially fulfills our second objective: providing a theoretical background for HOSD. The formalization is quite natural and provides some interesting insights toward a better understanding of OS discovery:

Assumptions: Simplifying assumptions made throughout the implementation of HOSD translate to properties in the underlying diagnosis model. It is then easier to be aware of those assumptions than with other OSD tools. It is also easier to understand the impact of relaxing a given assumption; by considering the corresponding diagnosis theory.

Complexity: We should expect the passive task of OS discovery to be NP-Complete if we consider a multiple fauts mode (several OSes can be associated to a single IP address, as it is the case with NATs for instance). In the single fault case, passive OSD is polynomial.

Moreover, we provided a polynomial algorithm for candidate generation for diagnosis tasks sharing the properties of OS discovery. This algorithm can now be used to implement the passive module of HOSD. This should provide a significant speed-up compared to the current ASP implementation.

Throughout this chapter, we addressed only one aspect of diagnosis: candidate generation. This corresponds to the passive module of HOSD. The other aspect deals with test selection in order to eliminate candidates. The study of candidate elimination will be done in Chapter 7. Candidate elimination corresponds to active OS discovery.

6.6 Contributions

This chapter provides three contributions:

- A formalization of operating system discovery as a diagnosis task.
- An explicit listing of the assumption we make to build our tool. These assumptions translate to properties of the selected diagnosis model.
- A polynomial algorithm for candidate generation for diagnosis tasks sharing the properties of operating system discovery.

6.7 Proposal

This chapter is mainly complete. The polynomial algorithm presented in Section 6.4 will be implemented in the passive module of HOSD and time benchmarks will be provided to measure the improvement over the ASP implementation.

Some interesting enhancements to the diagnosis modelization of HOSD are discussed in the Future Work Section 9.5.

Chapter 7

Diagnosis - Candidate Elimination for Active Operating System Discovery

In theory, there is no difference between theory and practice. In practice, there is.

-Yogi Berra

Abstract

In the previous chapter, we saw how diagnosis problem solving can be used to represent the operating system discovery task. The focus was entirely on the candidate generation part of diagnosis, corresponding to the passive module of OS discovery. Here we focus on the candidate elimination part of diagnosis which corresponds to active OSD. We first describe what is a test in the context of diagnosis. Then we extend the current diagnosis framework to provide more flexibility, based on the queries of a user. Finally, we study different test selection strategies to solve those queries.

7.1 Introduction

In the previous chapter, we presented a diagnosis framework which is completely static from the user point of view, see Section 6.2. Indeed, the user cannot interact with the diagnosis engine to obtain the knowledge he wants, or to orient the diagnosis process. In our OS discovery application, having multiple queries is important. As mentioned in Chapter 5, three queries are especially relevant to OSD: Single OS Query, Group OS Query and Exact OS Query. Thus it seems natural to extend the diagnosis framework to handle, at least, these queries.

To answer a specific query, the diagnosis engine must go through a test selection process. The execution of a test incurs a cost, thus we might want a selection strategy minimizing the total cost of the executed tests. On the other hand, we might need a very fast test selection strategy, so there is a classical tradeoff between the number of tests to execute (or, more generally, the cost of the solution) and the time spent selecting tests. As different diagnosis problems may have different requirements regarding that tradeoff, we believe a good diagnosis tool should allow the user to select

between different strategies. With this in mind, we propose to study different test selection strategies for three queries.

The rest of the chapter is structured as follows. To start, Section 7.2 proposes a formal characterization of a diagnosis test. Then, Section 7.3 extend the diagnosis framework presented in the previous chapter to handle queries. Section 7.4 presents the current approach proposed in the diagnosis literature for test selection. Afterwards, Section 7.5 discusses the properties that should be studied for each query. Section 7.6 provides a detailed study of test selection strategies for one specific query. At the end of the chapter we provide a short conclusion, a summary of the contributions made throughout this chapter, and the proposed work, related to the topic of this chapter, to be done during the thesis.

7.2 Test Representation

In the previous chapter, 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. In this section, we discuss the test representation we will be using in this chapter, mainly focusing on how to reason with tests. But first, let's 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. Finally, 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 tests generating malformed packets as being more costly than tests generating only well-formed packets. 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 and finally present our *outcome-based* representation.

7.2.1 Reiter's Test Representation

Reiter briefly discusses candidate elimination in [60]. 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 [60]). 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.

7.2.2 McIlraith's Test Representation

Most of the work around test representation for diagnosis has been conducted by McIlraith, see [5] [48]. In [47], 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.

This test representation seems quite general. However, it is not tailored towards reasoning for test selection. Below we provide a representation similar to this one, but directly oriented toward reasoning with tests.

7.2.3 Outcome-Based Test Representation

Our test representation is geared towards OS discovery and inspired for McIlraith's representation discussed above. A test in OS discovery consists of sending a stimulus to a computer and analyzing the response. We assume that tests can always be executed, so we do not consider pre/post conditions.

Definition 7.1 (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 7.1 (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 B.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 enabled and the TTL is 255. This is different from the behavior of MacOS, which would disable the DF bit and use a TTL of 64. ♢

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

Definition 7.2 (Possible Outcomes of a Test)
Given a test t, the set of possible outcomes of t, denoted O_t is

$$O_t = \{\Theta | \Theta \subseteq \mathit{OBS} \text{ and } \exists h \in \mathcal{H} \text{ such that } P_t(h) = \Theta\}$$

That is, $\Theta \subseteq \mathit{OBS}$ is a possible outcome of t if there is an hypothesis that will generate the observations Θ in response to t. ◯

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, several other factors must 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 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 7.2 (OSD test (continued from Example 7.1))
In Example 7.1, 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 $tcp(no, rstack, 64)$. ♢

Our approach is similar to the one proposed by McIlraith. However, by defining the outcomes of a test using observations, we make it easier to connect candidate elimination with candidate generation in a diagnosis engine. More importantly, our notion of prediction function makes the process of reasoning about tests much more intuitive, as we directly know which outcome to expect under a given hypothesis.

Note that we could extend our definition of a test to include preconditions as well. We omit this extension here, since it is not required for OS discovery.

7.3 Extending Diagnosis with Queries

In the previous chapter (see Figure 6.1) we presented the operation mode of a simple diagnosis tool. However, such a model has a fundamental restriction: the user cannot interact with the system by querying it. Since knowledge is the corner stone of diagnosis, it does not make sense to prevent the user from querying the knowledge base. The current literature about diagnosis considers a single static user goal: to

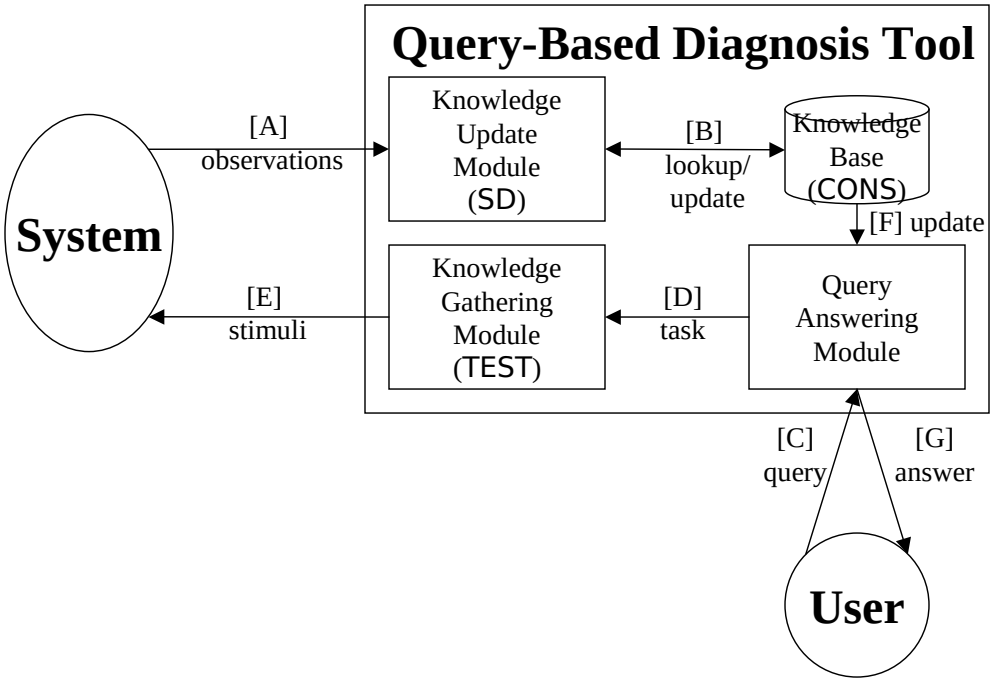


Figure 7.1: Query-Based Diagnosis Tool Behavior

find the actual diagnosis. We believe this lack of flexibility to be a drawback of the current diagnosis architecture, and we provide a query-based extension to give more flexibility to the user.

In what follows, we start by extending the diagnosis architecture to support user queries. Then we discuss three queries that are particularly interesting in the OS discovery domain. Moreover, we argue that these queries are generally meaningful, that is, they are interesting in other diagnosis domains. Finally, we illustrate the usefulness of our query-based diagnosis architecture.

7.3.1 Query-Based Diagnosis Architecture

Figure 7.1 presents an extended architecture for diagnosis. It is very similar to Figure 6.1, 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* with respect to the specific query.

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 details in the following sections.

7.3.2 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?

Actual Diagnosis Query: what is the actual diagnosis?

The Actual Diagnosis 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 Chapter 5 (see Section 5.2.2.1). In fact, the Single Candidate Query corresponds to the Single OS Query, the Group Candidate Query to the Group OS Query, and the Actual Diagnosis Query to the Exact OS Query.

However, for the query-based extension we propose to be adopted, the queries must also be meaningful in other diagnosis domains.

7.3.3 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.

7.3.3.1 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 important symptoms, the first step would be to rule out the possibility of the patient being contagious with a serious disease. This would amount to asking the query "Does the disease belong to the set of contagious serious 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.

7.3.3.2 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 to know why the clock is not working, we are solely interested to know 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.

7.3.3.3 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 Actual Diagnosis Query is usually more costly (i.e., requires more tests) then solving the Single Candidate Query. Below, we back that intuition.

7.3.4 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 Actual 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 Actual Diagnosis Query is generally harder (i.e., it requires more tests) then solving the Single Candidate Query, for instance.

To prove this assumption, we need to manipulate tests. As discussed in Section 7.2.3, 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 Actual Diagnosis Query is always a solution to any Single Candidate Query, see Proposition 7.1.

Proposition 7.1
Solving the Single Candidate Query for $h \in \mathcal{H}$ requires at most as many tests as solving the Actual Diagnosis Query.

Proof:
Assume the Actual Diagnosis 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 Actual Candidate Query, see Proposition 7.2.

Proposition 7.2

Solving the Actual Diagnosis 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 Actual Candidate Query. In the worst case, the first $n-2$ tests would refute only one hypothesis each, leaving still two candidates. Thus, solving the Actual Candidate Query can require $n-2$ more tests than solving the Single Candidate Query. □

7.4 Current Test Selection Strategy

Current diagnosis tools focus entirely on the objective of finding the actual diagnosis (i.e., they only consider the Actual Diagnosis Query). 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, see [27].

- The "one step lookahead" strategy employed by the greedy approach is *good enough*, see [26].

We disagree with these two claims.

For the first claim, we believe the tradeoff between computation time and test cost is domain dependant. 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 domains, executing extra tests might not be such a bad thing. In the medical domain, however, tests are extremely costly (in terms of execution 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 return 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.

Thus, we consider it is important to study other test selection strategies; both to provide flexibility to the user to choose from several strategies according to his needs and to have a good understanding of the underlying problem (e.g., solvability, tractability, etc.).

7.5 Query-Oriented Test Selection Strategies

Following our extension to diagnosis based on queries, we have to study the test selection problem individually for each query. Here, we provide a brief discussion regarding test selection for each of our three queries and we define the theoretical properties to consider for the different resulting problems.

7.5.1 The Queries

In the Actual Diagnosis Query, we simply want to know what is the actual diagnosis. A solution is a (possibly branching) sequence of tests such that after the execution of those tests we only have one candidate remaining, the actual diagnosis.

For the Group Candidate Query, we want to know if the actual diagnosis belongs to a set of diagnosis candidates $H \subseteq \mathcal{H}$. A solution is a (possibly branching) sequence of tests such that after the execution of the tests the resulting set of diagnosis candidates either contains only elements of H or contains no element of H.

For the Single Candidate Query, we want to know if the actual diagnosis is a specific hypothesis $h \in \mathcal{H}$. A solution is a (possibly branching) sequence of tests such that after the execution of those tests the resulting set of candidates either contains only h or does not contain h.

From the formulation of the problem above, we can clearly see that the reasoning process will focus on different things for different queries. For instance, a good strategy for the Actual Candidate Query seems to select tests that will refute as many candidates as possible. However, this strategy is not very good for the Single Candidate Query, where we prefer to concentrate on a given hypothesis. It also seems that the more information we have in the query, the more specific is our problem. Usually an under-specified problem is harder to solve as we cannot take advantage of the complete problem structure. These reasons provide an intuition why it is important to study test selection strategies for different queries. Below, we define the theoretical properties that we will study for the problem underlying each query.

7.5.2 Problem Properties

We propose to study the three problems, one per query, related to test selection. We study those problems with respect to four properties:

Optimal Characterizability: Is it possible to provide a formal definition of the optimal solution? In other words, given two solution, can we always tell which one is better?

Solvability: If we have a definition of the optimal solution, can we design an algorithm that will find it in finite time?

Tractability: If the problem is solvable, can we solve it in polynomial time or is it NP-Complete?

Approximability If the problem is intractable, is it approximable? That is, can we design a polynomial algorithm which provides a solution s such that the cost of s is no greater than some constant c times the cost of the optimal solution. In other words, regardless of the problem instance, we can find an approximate solution that is guaranteed to be "close" to the optimal.

As soon as the problem is optimally characterizable, we can compare a solution with the optimal one. As a result, we can evaluate approximation algorithms (or heuristics).

Based on our experience with the queries, our intuitions are that:

- the Single Candidate Query is intractable (but optimally characterizable and solvable) under an assumption regarding the structure of the tests.
- the other two queries are not even optimally characterizable.

If our intuitions are correct, then we have a formal justification to search for different algorithms for the different queries. We also have a formal justification for the existence of the Single Candidate Query (which is a special case of the Group Candidate Query).

We discuss below the early results we obtained concerning the Single Candidate Query.

7.6 Single Candidate Query

Here we study the Single Candidate Query with respect to the properties we just defined. We consider a single fault hypothesis space and we assume that all tests have an equal cost. We also assume that every test is *uniquely supporting*, see below.

Definition 7.3 (Uniquely Supporting Test)
A test t uniquely supports a hypothesis h if there is only one outcome $\Theta \in O_t$ such that $h \in \Gamma(\Theta)$ (recall that $\Gamma(\Theta)$ is the set of diagnosis candidates for Θ), i.e., t has only one outcome which does not refute h. A test is uniquely supporting if it uniquely supports every hypothesis. ◯

The uniquely supporting assumption holds in OS discovery as every OS has a single response to each test.

7.6.1 Problem Representation

We have a set TEST of available tests. Each test $t \in$ TEST is a partition of $\mathcal{H}$ (thanks to the uniquely supporting assumption). Thus, each test t can be represented by a family $\mathcal{S}^t$ of sets, each set representing the hypotheses that are mutually indistinguishable by t. Each test t is such that $\cap_{S \in \mathcal{S}^t} S = \emptyset$ (by the uniquely supporting assumption) and $\cup_{S \in \mathcal{S}^t} = \mathcal{H}$ (the prediction function is defined over $\mathcal{H}$). More formally, t is represented by the family:

$$\mathcal{S}^t = \{\Gamma(\Theta) | \exists h \in \mathcal{H} \text{ such that } P_t(h) = \Theta\}$$

In the Single Candidate Query we pick an element $h \in \mathcal{H}$ and want to know if h is the actual diagnosis or not. With respect to h, each test t can now be represented as a pair of sets $\langle S^t(h), S^t(\overline{h})\rangle$. Intuitively, $S^t(h)$ corresponds to the unique set of $\mathcal{S}^t$ (the family of sets defining t) containing h. On the other hand, $S^t(\overline{h})$ is the union of all sets defining t and not containing h. Formally,

- $S^t(h) = S$ such that $S \in \mathcal{S}^t$ and $h \in S$. Note that $S^t(h) = \Gamma(P_t(h))$.
- $S^t(\overline{h}) = \cup S$ such that $S \in \mathcal{S}^t$ and $h \notin S$. Note that $S^t(\overline{h}) = \mathcal{H} \setminus \Gamma(P_t(h))$.

Note that the pair $\langle S^t(h), S^t(\overline{h})\rangle$ still forms a partition (now a bipartition) of $\mathcal{H}$. See Figure 7.2.

Definition 7.4 (Single Candidate Query Solution)
A solution to the Single Candidate Query is a subset of tests $T \subseteq$ TEST such that $\cap_{t \in T} S^t(h) = \{h\}$; that is, the only hypothesis confirmed by all the outcomes confirming h is h itself. A solution is then a non-branching[1] *and unordered sequence of tests.*

[1] Which test to execute is independent of the result of the previous test.

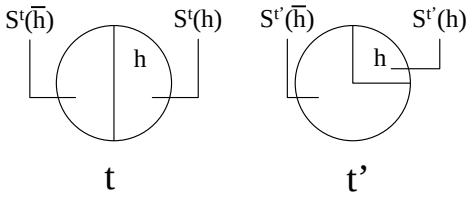


Figure 7.2: Graphical Test Representation

The size of a solution is the number of tests it uses. ◯

Note that the Single Candidate Query usually considers a current set of diagnosis candidates. However, our representation is equivalent in the following way. Given a set of candidates Γ, the universe would be $\mathcal{H} \cap \Gamma$ instead of $\mathcal{H}$. The sets defining a test would be $S_i^t \cap \Gamma$ instead of S_i^t.

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

Definition 7.5 (Discriminant Power)
Given a test t, a set of diagnosis candidates Γ and a specific hypothesis h, the discriminant power d of t for h with respect to Γ is the number of candidates from Γ that are eliminated by the outcome of t which confirms h. That is, $d = |\Gamma| - |\Gamma \cap S^t(h)|$.
◯

Proposition 7.3
Given a diagnosis problem , 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). □

7.6.2 Optimal Characterizability

Characterizing the optimal solution of the Single Candidate Query is straightforward. As mentioned above, a solution is a set of tests. The optimal solution is the solution with minimal cardinality.

If tests have different costs, the optimal solution is the solution for which the sum of the costs of its tests is the lowest.

7.6.3 Solvability

It is easy to see that the problem is solvable. Given the set of tests TEST, we can form at most $2^{|\mathsf{TEST}|}$ solutions. All we have to do to find the optimal solution is to build them all and then find the minimal one. This procedure, see Figure 7.3, clearly runs in $\Omega(2^{|\mathsf{TEST}|})$ in the worst case (simply consider step 2).

Figure 7.3: Brute Force Algorithm for Test Selection

BFTestSelection(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 $R \leftarrow \{c | \{c\} \in \Gamma\}$
2 FORALL $T \in \wp(TEST)$ in increasing size order
2.1 IF $\cap_{t \in T} S^t(h) \cap R = \{h\}$
2.1.1 RETURN T
2 RETURN "no solution"

7.6.4 Intractability

We've shown that the problem is solvable and we've provided a simple, but exponential, algorithm. We will show below, in Theorem 7.1, 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 $\mathcal{T}$ (corresponding to TEST in diagnosis), 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 $\langle \mathcal{H}, \mathcal{T}, h \rangle$ and an

integer j, find whether or not there exist a solution to the Single Candidate Query of size j or less. We call this problem SingleCandidateQueryD.

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 [35].

Definition 7.6 (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. ◯

7.6.4.1 NP-Completeness of 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 $T \subseteq \mathcal{T}$, 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 $|\mathsf{CONST}|$).

Theorem 7.1 (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 $\mathcal{T}$, and 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 $\langle \mathcal{H}, \mathcal{T}, h \rangle$ has a solution to the Single Candidate Query of size j or less.*

Reduction:

- $j = k$
- $h =$ *"h", an element not in* $\mathcal{U}$
- $\mathcal{H} = \mathcal{U} \cup \{h\}$
- $\mathcal{T}$*: 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.

$\Rightarrow$[2]*:*

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* $\langle \mathcal{H}, \mathcal{T}, h \rangle$*. It has size no greater than* j *as* $|T| = |C| = k' \leq k = j$*. So let's explain why* T *is a solution to the Single Candidate Query. Since* C *is a set cover for* $\mathcal{U}$*, we now 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's denote this set by* $X(e)$*. But, by the definition of* $\mathcal{T}$*, 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* $\cap_{t \in T} S^t(h) = \{h\}$*.*

$\Leftarrow$[3]*:*

Assume $\langle \mathcal{H}, \mathcal{T}, h \rangle$ *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 let's 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* $\mathcal{U}$*.* □

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

[3]If $\langle \mathcal{H}, \mathcal{T}, h \rangle$ 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.

7.6.5 (In)Approximability

We currently do not know whether the problem is approximable or not. The first step towards answering that question is to check the status of the greedy algorithm.

Given the Single Candidate Query for $h \in \mathcal{H}$, the greedy approach would select the test t with the maximal discriminant power, see Figure 7.4. This algorithm has a worst case time complexity[4] 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 7.3, the algorithm of Figure 7.4 has a worst case complexity of $O(|\mathsf{TEST}|^2 \times |\mathsf{CONST}|^2)$.

Example 7.3 illustrates that the greedy algorithm is an unbounded approximation.

Example 7.3 (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:

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

[4]It might seem strange that the complexity of the algorithm depends on $|\mathsf{CONST}|$, since this is not a parameter. However, recall that our representation of a test, see Section 7.6.1, is such that a test is a bipartition of CONST.

Figure 7.4: Greedy Algorithm for Test Selection

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 $t \leftarrow$ getAny(TEST)
2.3 FORALL $t' \in \mathsf{TEST}$
2.3.1 IF discrimPower(t',Γ,h) > discrimPower(t,Γ,h)
2.3.1.1 $t \leftarrow t'$
2.4 IF discrimPower(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

---

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 7.1); assume[5] 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 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 the $k = 1$ tests. $\diamondsuit$

The key idea of the above example is that increasing the value of k by one (adding

[5]We could easily build an example where t_a and t_b never have the highest discriminant power until they are the only tests left. It would simply be longer and more tedious.

Table 7.1: Discriminant Power

Tests \ Candidates	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

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 the greedy algorithm provide an unbounded approximation of the optimal solution.

7.6.6 A Spectrum of Test Selection Strategies

So far, we have discussed two algorithms to solve the Single Candidate Query. In Section 7.6.3 we provided an exponential, $O(2^{|\mathsf{TEST}|})$, brute force algorithm which is guaranteed to get the optimal solution. Then, in Section 7.6.5 we presented a polynomial, $O(|\mathsf{TEST}|^2 \times |\mathsf{CONST}|^2)$, greedy algorithm which is suboptimal in an unbounded way. We can easily define another algorithm that would pick a test arbitrarily, execute it, and loop until it can answer the query, see Figure 7.5. This algorithm runs in $O(|\mathsf{TEST}|)$ and will, on average, be worst than the greedy one with respect to the solution returned.

Based on these three algorithms, Figure 7.6 represents the spectrum of the tradeoff between computation time and solution quality. The main question is whether there is an algorithm to fill the gap between the greedy and the brute force algorithms. What we want is a polynomial algorithm providing a bounded approximation of the optimal solution, if such an algorithm exists.

Figure 7.5: Random Algorithm for Test Selection

ArbitraryTestSelection(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 $t \leftarrow$ getAny(TEST)
2.3 $T = T \cup t$
2.4 $\mathsf{TEST} = \mathsf{TEST} \setminus \{t\}$
2.5 $\Gamma = \Gamma \cap S^t(h)$
3 RETURN T

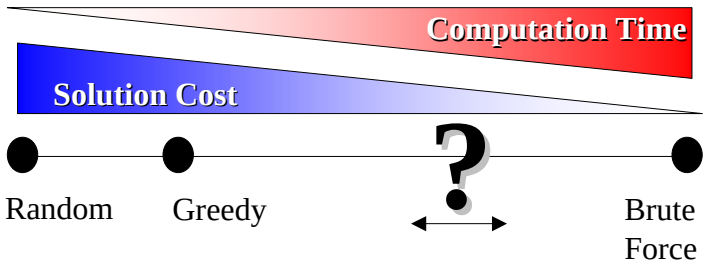


Figure 7.6: Spectrum of Algorithms for the Single Candidate Query

7.7 Conclusion

Driven by our quest to build our tool using a strong theoretical background, we proposed an extension to the current diagnosis theory. This extension is based on user queries and provide more flexibility to the user. Although this extension was driven by the OS discovery problem, it can also be useful in several other diagnosis domains, like medical and engineering diagnosis.

Our extension to query-based diagnosis creates the test selection problem, i.e., what is the best test to execute to answer a specific query in a given situation. We defined the theory of test selection in diagnosis. Unlike the current work on test

selection, which only considers the greedy approach to solve the Actual Diagnosis Query, we propose a spectrum of algorithms in the tradeoff between computation time and solution quality.

Since each query has its own characteristics, a single algorithm will probably not be adequate for all of them. Moreover, if it turns out that the problems have different theoretical properties, then this will be an indication that we should not try to achieve the same level of solution quality when solving them. Hence, we study the test selection strategies individually for each query.

7.8 Contributions

This chapter provides three main contributions:

- A representation of diagnosis tests which eases the act of reasoning about possible outcomes during the test selection process.
- A Query-Based approach to diagnosis. Asking the right query can drastically reduce the cost incurred to obtain an answer.
- Establishing the theoretical properties for the Single Candidate Query (under the assumption that tests are uniquely supporting):
  - It is optimally characterizable.
  - It is solvable.
  - It is intractable (i.e., it is NP-Hard).
  - The greedy approach provides an unbounded approximation of the optimal solution.

Accepted and/or pending publications with respect to this chapter are:

- [15]: 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.

- [16]: François Gagnon and Babak Esfandiari. A Query-Based Approach for Test Selection in Diagnosis - Operating System Discovery as a Case Study. Submitted to Artificial Intelligence Review Journal - Special Issue on AI & Pervasive Computing, 2008.

7.9 Proposal

To complete the work of this chapter, we propose to:

- Look for a bounded approximation algorithm for the Single Candidate Query or demonstrate the inaproximability of the problem.
- Study the theoretical properties of the other two queries. Our intuition is that they are not even optimally characterizable. This could provide a strong reason why a greedy approach should be used (unlike the current claim that computation time is prohibitive with respect to test costs).
- Design several algorithms for each of the queries to fill the spectrum created by the tradeoff computation time vs solution cost.
- Provide an empirical study of the different algorithms for each query. Because an algorithm is exponential in the worst case does not mean it cannot be used in a specific context. And because an approximation algorithm is unbounded does not mean it won't provide satisfying results in a specific application. We will use OS discovery with the three queries for this empirical study.

Chapter 8

VNEC - A Virtual Network Experiment Controller

Efficiencies are borne of laziness.
-Lori McGurran

Abstract

In the previous chapters, we have discussed how to build a knowledge-oriented OS discovery tool. So far, we have assumed that we know how each operating system behave in a specific situation. However, acquiring this knowledge is not an easy task. Most OS discovery tools rely on users to submit new fingerprints in a ad hoc manner. Here, we discuss a tool allowing to automatically collect traffic traces from which OS fingerprints can automatically be extracted.

8.1 Introduction

The third objective of this thesis is to provide a systematic and automated way of gathering OS fingerprints. 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, 77]) 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 Chapter 3, 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 and cannot be triggered remotely on-demand. Other tests are based on reactive events, i.e., events occurring in response to a stimulus. Since reactive tests do not require any actions from the fingerprinted VM, we simply need to stimulate it remotely. Spontaneous tests, on the other hand, must be initiated by the fingerprinted VM itself. Thus we have to be able to control[1] every VM in order to force them to generate those spontaneous events.

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

In this chapter, we present VNEC [12] (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`. The rest of the chapter is structured as follows. Section 8.2 describes a generalized version of the OS fingerprint gathering problem. Then, Section 8.3 briefly discusses some related work with an emphasis on the CRC virtual environment used in Chapter 2. Section 8.4 presents VNEC and Section 8.5 explains how it can be used for OS fingerprint gathering. Finally, we provide a short conclusion and a summary of the contributions made throughout this chapter.

8.2 The General Problem

OS fingerprinting is one type of experiments that can be executed in a virtual environment. The benefits are that we can study several 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:

- Studying the spreading patterns of viruses.
- Analyzing the behavior of different targets with respect to some given attacks. This experiment could gather data to use when monitoring the attack side effect as IDS context, see Section 2.3.1.4 of Chapter 2

We want VNEC to be able to support these experiments, so it is not only used by us for OS fingerprinting.

Performing these experiments in a physical network would be time consuming, since the computers need to be cleaned after each virus, 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 effect of security sensitive experiments do not spread to the physical machine hosting the experiment.
- It has to provide a wide variety of guest OSes.

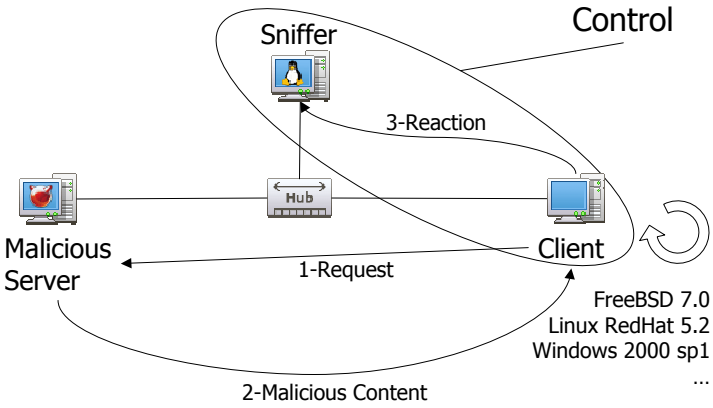


Figure 8.1: Client-to-Server Attack Experiment

- The environment must be able to control every VM to be able to generate spontaneous events (for OS fingerprinting) and client-to-server attacks (for attack reaction study, see Example 8.1).

Example 8.1 (client-to-server attack scenario in a virtual environment) Figure 8.1 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 several 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.

8.3 Related Work

Before building the tool, we had to settle on a specific virtualization technology to use as the virtual environment. The choice was made easy by our requirement to have a wide variety of OSes. We choose to use VMWare workstation [75] as it supports Windows, Linux (Red Hat, Mandrake, Fedora Core, Ubuntu, etc.), FreeBSD, NetBSD, OpenBSD, Novel NetWare, etc. Alternative technologies included VirtualPC [9], QEMU and Xen [38] (Chapter 14), UML (User Mode Linux) [29], and Basilisk [6], but they typically support much fewer guest OSes.

Some tools to "control" virtual machines already exist. Of particular interest are the following: VMWare VIX API [76], HP SmartFrog project [53], the VNUML tool [44], and the CRC virtual environment.

VIX is an API developed by VMWare to provide control over the virtual machines (e.g., power up, file copying, command execution). However, the VIX functions allowing communication with a VM work only if the the VM is running a recent Linux or Windows OS[2]. This is a major limitation, since task cannot be dispatched to all virtual machines.

The SmartFrog project is a general framework to manage distributed systems. To use VMWare technology, SmartFrog uses a Java wrapper on VIX, thus it can act as a controller for virtual networks. However, since SmartFrog relies entirely on VIX, it inherits its limitations and can only dispatch tasks to Linux and Windows VMs as well.

The VNUML tool (Virtual Network User Mode Linux) is a controller for the UML virtualization technology. However, since UML supports only Linux guest OSes, VNUML is even more limited than SmartFrog. Moreover, VNUML cannot dispatch tasks to the VMs. Its control over virtual machines is limited to power on, shut down, network configuration, and the like.

VNEC is an extension of a tool developed at the Communication Research Center (CRC) [20] in Canada, see Section 8.3.1. This earlier version could only dispatch commands to Windows and Linux virtual machines. The need to dispatch commands

[2]These functions requires the VMWare tools to be installed on the VM, and only recent Linux or Windows VM can install them.

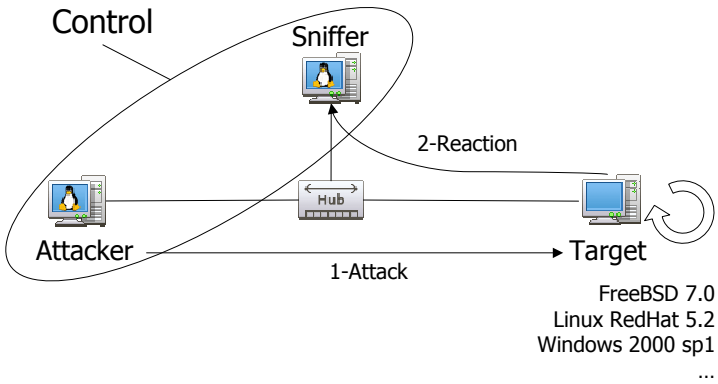


Figure 8.2: Server-to-Client Attack Experiment

to all virtual machines was the principal motivation to develop VNEC.

There could be other undocumented ad hoc solutions to the creation of virtual network environments. However, we are not aware of a solution allowing to control VMs that would work for most guest operating systems.

Since VNEC uses VMWare, every OS supported as a guest by VMWare can be used in the experiments. However, unlike VIX, VNEC is able to dispatch commands to every virtual machine not just to those running Linux or Windows. This is important for many data gathering experiments such as studying the reaction of different OSes against a given attack and gathering OS fingerprints.

8.3.1 CRC Virtual Environment

The CRC environment can only execute experiment similar to the one in Example 8.2. Here, only two VMs needs to be controlled and they do not change from one run to the other. What changes is the target VM, which ranges over several different OSes and does not need to execute any task.

Example 8.2 (server-to-client attack scenario in a virtual environment) Figure 8.2 illustrates the scenario of a server-to-client attack (in a virtual environment). The attacker first attacks the target (step 1). Then, the sniffer records the traffic generated by the client as a reaction to the attack (step 2). We want to replay this experiment several times using different target VMs to see how different OSes react to the same attack. This scenario is easier than the one presented in Example

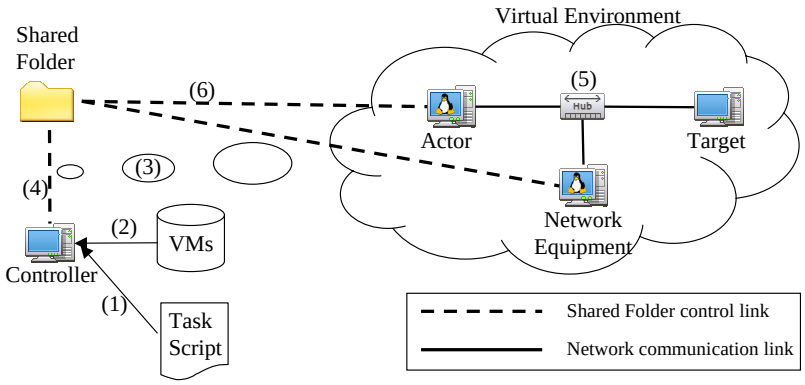


Figure 8.3: CRC Virtual Environment Architecture

8.1 because only the two static VMs (the attacker and the sniffer) need to execute tasks. ♢

The example above can also be seen from our OS fingerprinting point of view. Instead of an attacker, we talk about a stimulator, and instead of attacking the target, it generates a stimulus to trigger a response from the target. This would only allows to gather fingerprints for reactive tests.

The virtual environment at CRC works as follows (according to Figure 8.3):

- The user provides an XML script to the experiment controller (step 1 in Figure 8.3). The script contains the VM to be used as a target and the task to be executed by the actor.
- The controller fetches the virtual machines from a VM repository (step 2) and then creates the virtual network required for the experiment (step 3).
- Once the network is ready, the controller informs the actor of the task he should execute (step 4). Since the actor never changes, it's a Linux VM, the task is dispatched through a VMWare shared folder.
- The actor executes the task (step 5) and then sends the result back to the controller (step 6).

So far, we have been using the CRC virtual environment for fingerprint gathering. Reactive fingerprints can be gathered automatically using this environment. For the

spontaneous fingerprints, we have been gathering them manually which is a tedious, error-prone, and not easily repeatable process.

The main limitation of this architecture is the inability to dispatch commands to all virtual machines. As a result, client-to-server attack scenarios as well as spontaneous event fingerprints scenarios cannot be performed. VNEC has been developed mainly to address this limitation.

8.4 VNEC Architecture

VNEC has three modules, each one is detailed in its own section below:

- Network specification. The network topology and VMs to use.
- Task workflow specification. The tasks to be executed by the VMs and their order.
- Experiment execution. Configuring the VMs and dispatching the tasks in the desired order.

8.4.1 Network Specification

Figure 8.4 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.
- Specifying the network topology by connecting the components according to some rules (e.g., computers cannot connect to computers, each computer has at most one connection).
- Associating each computer to a virtual machine (from a set of pre-existing VMs).

In Figure 8.4, the user connected five machines (in clockwise order from top-right: FreeBSD NetBSD, OpenBSD, Linux, and Windows) using two hubs and a router.

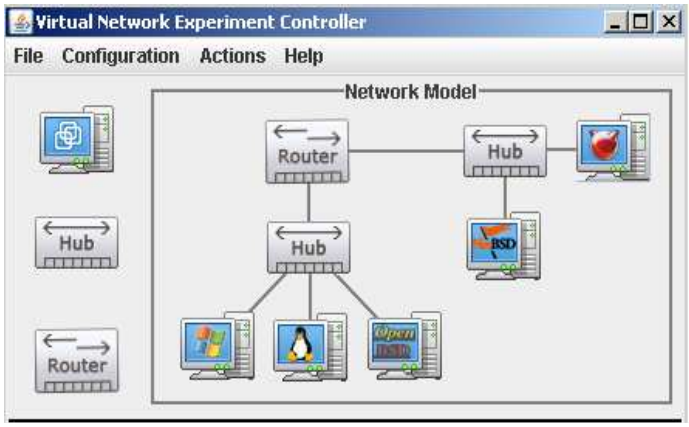


Figure 8.4: Snapshot of VNEC - Network Specification

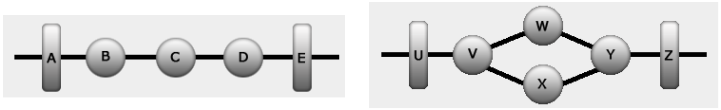
The network consists of two segments: FreeBSD with NetBSD is one and the other three hosts form the other. Routers are yet to be implemented, using dedicated VMs, while hubs are the default behavior on a VMWare VMNet. By clicking on a computer icon, it is possible to select the snapshot to use for the specific VM. By default, the current snapshot is used.

8.4.2 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 [10] with a single source and a single sink[3] where each node corresponds to a task (Figure 8.5). 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

[3]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 8.5: Examples of Task Workflows

some parameters; for instance, the delete file command structure 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`" would translate to "`rm -f name`" on a Linux VM and to "`del name`" 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 all the virtual machines in the network. For instance, the task workflow shown in Figure 8.5(a) starts with task A which is executed by all VM (say power on). Once task A is completed, task B, is performed by a single VM (say Linux starts recording traffic). Afterwards, task C (say OpenBSD launches a specific attack against Windows) and task D (say Linux stops recording the traffic) are executed in sequence. Finally, E (power off) is applied on all virtual machines.

A task workflow does not have to be linear as displayed in Figure 8.5(b). Once task V is completed, both tasks W and X begin concurrently. Task Y will start only after both W and X are completed.

8.4.3 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 the virtual machines: 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 place it on the shared folder to be processed by the corresponding VM. This process is both simple and safe[4]. However, the shared folder feature is available only for some virtual machines[5]. 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 8.6. 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.

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 8.6, VNEC works as follows:

- The user provides an experiment which consists of a network and task workflow specification (step 1 of Figure 8.6).
- The controller fetches the specified VMs from a repository (step 2) and then creates the virtual networks (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.

[4]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.

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

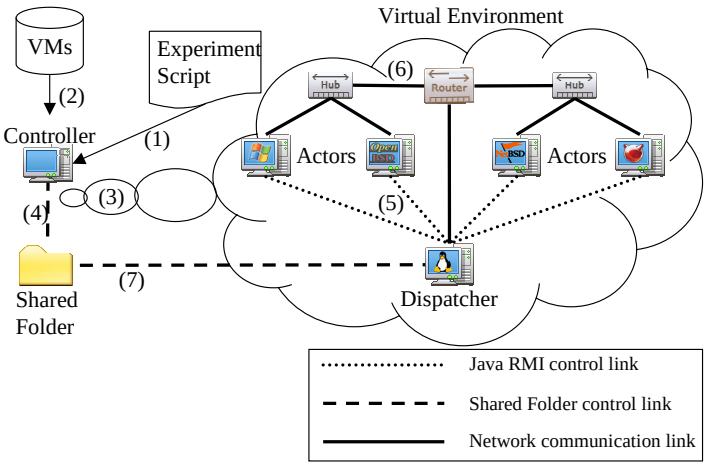


Figure 8.6: VNEC Communication Architecture

- 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. It is also general enough to be used for other network experiments, not just for OS fingerprinting.

8.5 OSD 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.

8.5.1 Reactive Events

We first consider how to use VNEC to gather fingerprints for the TCP RstAck test (see Definition B.9). We are interested about 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 yes, no 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 = yes, TTL = 64
- DF = no, 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.
- Sniffer starts recording traffic.
- Stimulator sends the two stimuli (on a closed port).
- 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 OSes used in the experiment.

8.5.2 Spontaneous Events

Now let's consider how to use VNEC to gather fingerprints for the TCP Syn test (see Definition B.1). We are interested about 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 ways to get a machine to send a SYN packet: open a FTP connection, open a web page, initiate a SSH connection, open a telnet connection, etc. We use FTP, SSH and telnet for all VMs. This allows us to obtain a significant 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.
- Sniffer starts recording traffic.
- Fingerprinted host tries to open a FTP connection on dummy target.
- Fingerprinted host tries to open a SSH connection on dummy target.
- Fingerprinted host tries to open a telnet connection on dummy target.
- Sniffer stops recording traffic.
- Retrieve the traffic trace from the sniffer.
- Power off all VMs.
- Start over with a different fingerprinted host.

8.5.3 Limitation

The main limitation of our approach for gathering fingerprints is not 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, several networking devices (e.g., switches, printers, handheld devices, game consoles) run a 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 (it is limited to fingerprinting OSes only). Network devices are

important from a security point of view because, like OSes, they suffer from vulnerability (e.g., SecurityFocus BID 31092 lists 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 tool include some firwares in their database.

Since our fingerprint gathering technique does not work for firmware, we currently have to rely on the same ad hoc process of manually collecting fingerprints for firmware.

8.6 Conclusion

VNEC is a generic and powerful tool. It can handle several types of experiments. Moreover, it can use a wide variety of guest operating systems thanks to VMWare virtualization technology, and it can dispatch commands to all virtual machines. This makes VNEC a perfect tool for OS fingerprinting experiments. Although VNEC is developed with a specific application in mind, i.e., data collection for OS fingerprinting, it will be helpful in several other domains, e.g., virus propagation and attack reaction behavior.

VNEC allows systematic and automatic data collection for OS fingerprinting. This fulfills the third requirement of this thesis: enhance the OS fingerprint collection process with respect to other OS discovery tools.

8.7 Contributions

The main contribution of this chapter is an open source tool, VNEC, that automatically executes network experiments in a virtual environment. We believe VNEC to be general enough to be used for several types of network experiments, not just OS fingerprinting. VNEC can be downloaded from `vnec.sourceforge.net`.

The following paper concerning VNEC has been published:

- [14]: 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.

8.8 Proposal

Concerning the topic presented in this chapter, we propose to define an XML language for experiment specification and implement save/load functionalities in VNEC. This will allow users to use XML as a scripting language for VNEC, instead of using the GUI. In doing so, we make the experimentation process even more automated.

Other interesting extensions are discussed as Future Work in Section 9.5.

Chapter 9

Proposal Summary

Prediction is very difficult, especially if it's about the future.
-Niels Bohr

This chapter provides an overview of the thesis proposal. It first summarizes the document and provides the main conclusions obtained so far. Then, it highlights the remaining deliverables and provides an estimated timeline for their completion, see Section 9.3. It then summarizes the contributions made so far, see Section 9.4. Finally, it provides some interesting ideas regarding possible ares for future work, see Section 9.5.

9.1 Thesis Summary

First, Chapter 2 measured the potential of using information about the target configuration for filtering non-critical alarms in IDSes. The results we extremely promising. Moreover, it also measured how good current OSD tools would be to gather that information and demonstrated that they were not good enough.

Then, Chapter 3 studied the classical approaches and tools to OSD and discussed several limitations explaining their inability to accurately gather IDS context information. Based on that, Chapter 5 proposed a hybrid approach to OSD based on knowledge management. Moreover, it presented a partial implementation of our tool, called HOSD, that already looks very promising. However, HOSD cannot be used in real-time applications yet due to excessive computation times.

To obtain complexity results about our problem in order to optimize HOSD, Chapter 6 and 7 formalized OSD as a diagnosis task. Chapter 6 proposed a fast algorithm for the passive module of HOSD based on the candidate elimination algorithms provided by diagnosis theory. Then Chapter 7 extended the current diagnosis framework

with a query-based approach to provide more flexibility to the user. It also studied partially the test selection problem for candidate elimination in diagnosis. Once the study about test selection is completed, this will provide the algorithms needed to implement the active module of HOSD.

Finally, Chapter 8 presented VNEC, a virtual network environment allowing us to gather OS fingerprints in a systematic and automated way.

9.2 Conclusion

Based on the results obtained so far, we believe the following key points are worth mentioning:

- Target configuration information is extremely relevant for IDS context, as it can filter out a significant amount of non-critical alarms (75% in our experiment).
- Current OSD tools, however, are not adequate for IDS context gathering; achieving only 1/3 of their potential. The following reasons explain why current tools are not good for context gathering:
  - Passive tools do not memorize past events nor previous deductions.
  - Active tools are intrusive and thus 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; at least for passive OSD.
- Although answer set programming is very convenient to implement passive OSD, its time complexity makes it unusable for implementing a real-world OSD tool.
- Diagnosis theory has proven to be a very natural and useful formalization of the OS discovery problem.
- 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.

9.3 Proposal

Regarding the theoretical aspect of the active module for HOSD (i.e., the content of Chapter 7), several improvements remain to be done:

1- Determine the theoretical properties of the remaining two queries (i.e., the Group Candidate Query and the Actual Diagnosis Query).

2- Establish the inapproximability of the Single Candidate Query or provide a bounded approximation algorithm.

3- Design and implement several algorithms for each query (to fill the spectrum created by the tradeoff between computation time and solution quality).

4- Benchmark those algorithms (in terms of time and solution quality) using the OS discovery application.

With respect to our HOSD tool (presented in Chapter 5), we will:

5- Re-implement the passive module based on the fast candidate generation algorithm provided in Section 6.4.

6- Benchmark the new passive module implementation for computation time.

7- Implement the active module based on the test selection theory of diagnosis.

8- Compare the full HOSD implementation with other tools. We expect to be significantly better than every other existing tool.

9- Create a new experiment to test the Single OS Query (experiments for the Group OS Query and the Exact OS Query are already defined).

Finally, to make our virtual environment really automated, we will:

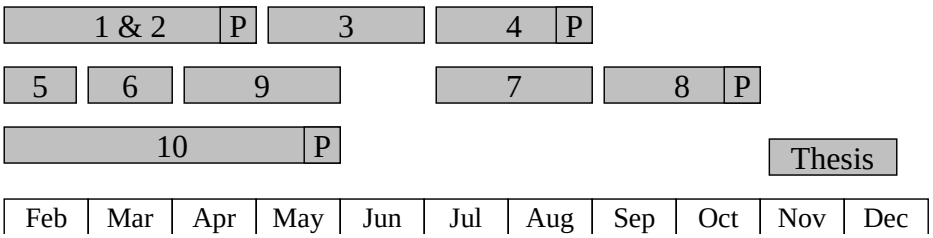


Figure 9.1: Tentative Timeline

10- Define an XML-based scripting language to specify an experiment and allow the user to run an experiment by providing its script.

Figure 9.1 illustrates our expected timeline with respect to the tasks mentioned above[1]. We expect the work proposed to be completed by the end of 2008.

9.4 Contributions

The results obtained so far provide several contributions across different fields:

- Establishing the effectiveness of target configuration information as IDS context.
- Proposing a better approach to operating system discovery based on solid theoretical grounds.
- Extending diagnosis theory with a query-based approach.
- Designing a framework to automatically execute network experiments in a virtual environment.

9.4.1 Publications

Among the contributions we've made so far, we have published a few papers: [14], [15], and [18]. We also have two pending submissions for journal publications: [17] and [16]. We also have a paper ready to be submitted [19] and we expect several other publications before the end of this thesis.

[1]Each occurrence of a "P" in the diagram represents an expected publication.

9.4.2 Tools

Moreover, we have developed two open source tools:

- HOSD. A hybrid tool for operating system discovery. Prototype available from `hosd.sourceforge.net`.
- VNEC. A tool to specify and execute network experiments in a virtual environment. Available in pre-alpha release from `vnec.sourceforge.net`.

9.5 Future Work

The work presented in this thesis could also be extended in other interesting ways. For instance:

- Concerning the use of contextual information to eliminate non-critical alarms in IDSes, an experiment comparing the different contextual approaches could be run. The goal is to see if the approaches can complement each other.
- The diagnosis model representing HOSD could be enhanced in the following ways:
  - Adopting a multiple faults model to handle NATs.
  - Relying on prior probabilities to take into account the popularity of different OSes.
- It would be interesting to generalize the test selection theory to other diagnosis domains, i.e., with different properties and constraints than OSD.
- With respect to our virtual environment VNEC, the following extensions would make the tool more suitable for generic network experiments:
  - Extend the architecture to support the distribution of an experiment across several physical machines. This will allow us to run experiments requiring large networks.

  - Extend the architecture to support multiple virtualization technologies. The objective is to be able to build a network with VMs from two different technology in such a way that the VMs can still communicate with one another. This will give us access to a wider variety of OSes.

- Integrate HOSD in an existing IDS to provide real-time filtering of non-critical alarms.

Appendix A

IDS Context Experiment Detailed Information

A.1 Dataset Information

Table A.1: Dataset BID and Exploits

BID	Exploit	BID	Exploit
1163	RFParalyze.c	7116	rs_iis.c
1331	crash_winlogon.c	7116	wd.pl
1578	iis_source_dumper.pm	7116	Xnuxer.c
1806	ALL_UNIEXP.C	7230	bysin2.c
1806	iisuni.c	7294	0x333hate.c
1806	iis-zang.c	7294	0x82-Remote.54AAb4.xpl.c
1806	unicodecheck.pl	7294	samba_exp2.tar.gz
1806	unicodexecute2.pl	7294	samba_trans2open.pm
2010	winnuke._eci.c	7294	sambal.c
2010	winnuke.pl	8035	iis_nsiislog_post.pm
2124	7350oftpd.tar.gz	8205	0x82-dcomrpc_usemgret.c
2417	solaris_snmpxdmid.pm	8205	30.07.03.dcom.c
2503	apache2.pl	8205	dcom.c
2674	iis_printer_bof.c	8205	DComExpl_UnixWin32.zip
2674	iis50_printer_overflow.pm	8205	msrpc_dcom_ms03_026.pm
2674	iis5hack.pl	8205	oc192-dcom.c
2674	iiswebexplt.pl	8205	rpc!exec.c
2674	jill.c	8315	0x82-WOOoou Happy_new.c
2674	sol2k.c	8315	0x82-wu262.c
2708	decodecheck.pl	8459	MS03-039-linux.c

Continued on next page

Table A.1 – Dataset BID and Exploits

BID	**Exploit**	**BID**	**Exploit**
2708	decodexecute.pl	8615	solaris_sadmind_exec.pm
2708	execiis.c	8826	MS03-04.W2kFR.c
2708	IIS_escape_test.sh	8826	ms03-043.c
2708	Iisenc.zip	9633	msasn1_ms04_007_killbill.pm
2708	iisex.c	9635	MS04-007-dos.c
2708	iisrules.pl	9751	servu_mdtm_overflow.pm
2708	iisrulessh.pl	10078	warftpd_165_pass.pm
2708	lala.c	10078	warftpd_165_user.pm
2906	fpse2000ex.c	10108	HOD-ms04011-lsasrv-expl.c
3064	zp-exp-telnetd.c	10108	lsass_ms04_011.pm
307	iis40_htr.pm	5556	smbnuke.c
3335	m00-apache-w00t.c	10108	win_msrpc_lsass_ms04-11_Ex.c
3581	bid3581.txt	10115	IIS5.0_SSL.c
3581	ftpglob.nasl	10115	sslbomb.c
4006	msdtc_dos.nasl	10116	THCIISSLame.c
4482	msftp_dos.pl	10116	windows_ssl_pct.pm
4482	msftp_fuzz.pl	11372	HOD-ms04031-expl.c
4485	DDK-IIS.c	11763	wins.c
5033	apache_chunked_win32.pm	11763	wins_ms04_045.pm
514	kod.c	6005	MultiWinNuke.c
514	kox.c	6005	winnuke.c
514	pimp.c	7106	samba_nttrans.pm
529	msadc.pl	7106	sambash.c
5311	mssql2000_resolution.pm	11820	iis_w3who_overflow.pm
5411	mssql2000_preauthentication.pm	14513	ms05_039_pnp.pm
754	rfpoison.py	7116	linux-wb.c

Table A.2: Dataset Operating Systems

Operating System	**Operating System**
FreeBSD 4.0	Linux Fedora 1 Server
FreeBSD 4.1	Linux Fedora 2 Server
FreeBSD 4.1.1	Linux Fedora 3 Server
FreeBSD 4.10	Linux Fedora 4 Server
FreeBSD 4.11	Linux RedHat 6.0
FreeBSD 4.2	Linux RedHat 6.1
FreeBSD 4.3	Linux RedHat 6.2
FreeBSD 4.4	Linux RedHat 7.0 Server
FreeBSD 4.5	Linux RedHat 7.1 Server
FreeBSD 4.6	Linux RedHat 7.2 Server
FreeBSD 4.6.2	Linux RedHat 7.3 Server
FreeBSD 4.7	Linux RedHat 8.0
FreeBSD 4.8	Linux RedHat 9.0
FreeBSD 4.9	Linux SuSe 10.0
FreeBSD 5.0	Linux SuSe 7.0
FreeBSD 5.1	Linux SuSe 7.1
FreeBSD 5.2	Linux SuSe 7.2
FreeBSD 5.3	Linux SuSe 7.3
FreeBSD 5.4	Linux SuSe 8.0
NetBSD 1.5.1	Linux SuSe 8.1
NetBSD 1.5.3	Linux SuSe 8.2
NetBSD 1.6	Linux SuSe 9.0
NetBSD 1.6.1	Linux SuSe 9.1
NetBSD 1.6.2	Linux SuSe 9.2
OpenBSD 2.6	Linux SuSe 9.3
OpenBSD 2.7	Windows 2000 std Server
OpenBSD 2.8	Windows 2000 std Workstation

Continued on next page

Table A.2 – Dataset Operating Systems

Operating System	**Operating System**
OpenBSD 2.9	Windows 2003 std Server
OpenBSD 3.0	Windows 2003 std sp1 Server
OpenBSD 3.1	Windows 95
OpenBSD 3.2	Windows 98 SE Workstation
OpenBSD 3.3	Windows 98 std Workstation
OpenBSD 3.4	Windows Millenium std Workstation
OpenBSD 3.5	Windows NT 4 sp3 Server
Windows 2000 sp1 Server	Windows XP Home sp1a Workstation
Windows 2000 sp1 Workstation	Windows XP Home sp2 Workstation
Windows 2000 sp2 Server	Windows XP Home Workstation
Windows 2000 sp2 Workstation	Windows XP Professional sp1 Workstation
Windows 2000 sp3 Server	Windows XP Professional sp2 Workstation
Windows 2000 sp3 Workstation	Windows XP Professional Workstation
Windows 2000 sp4 Workstation	

A.2 Precision/Recall Details for Classical OSD Tools

Table A.3: Precision/Recall Details for OSD Tool ettercap (passive)

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	44/ 44	44/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	15/ 15	15/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	15/ 15	15/ 172
2417	56/ 56	56/ 70
2503	0/ 0	0/ 33
2674	90/ 90	90/ 194
2708	0/ 0	0/ 261
2906	15/ 15	15/ 32
307	15/ 15	15/ 35
3335	21/ 26	21/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133

Continued on next page

Table A.3 – Precision/Recall Details - continued from previous page

Measure / **BID**	**Precision**	**Recall**
529	0/ 0	0/ 28
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	45/ 45	45/ 53
8035	71/ 71	71/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.4: Precision/Recall Details for OSD Tool Nmap

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	22/ 23	22/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	8/ 8	8/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	6/ 6	6/ 172
2417	20/ 20	20/ 70
2503	0/ 0	0/ 33
2674	36/ 36	36/ 194
2708	0/ 0	0/ 261
2906	6/ 6	6/ 32
307	6/ 6	6/ 35
3335	16/ 16	16/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	61/ 64	61/ 133
529	0/ 0	0/ 28
5556	0/ 0	0/ 12

Continued on next page

Table A.4 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	23/ 23	23/ 53
8035	24/ 24	24/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	2/ 2	2/ 23
8615	0/ 0	0/ 35
8826	2/ 2	2/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.5: Precision/Recall Details for OSD Tool p0f (RstAck)

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	0/ 0	0/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	0/ 0	0/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	60/ 60	60/ 70
2503	0/ 0	0/ 33
2674	16/ 16	16/ 194
2708	0/ 0	0/ 261
2906	0/ 0	0/ 32
307	16/ 16	16/ 35
3335	0/ 0	0/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.5 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	0/ 0	0/ 53
8035	64/ 64	64/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.6: Precision/Recall Details for OSD Tool p0f (StrayAck)

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	3/ 5	3/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	0/ 0	0/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	10/ 10	10/ 70
2503	0/ 0	0/ 33
2674	0/ 0	0/ 194
2708	0/ 0	0/ 261
2906	1/ 1	1/ 32
307	0/ 0	0/ 35
3335	0/ 0	0/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.6 – Precision/Recall Details - continued from previous page

Measure / **BID**	**Precision**	**Recall**
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	2/ 2	2/ 53
8035	13/ 13	13/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.7: Precision/Recall Details for OSD Tool p0f (Syn)

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	0/ 0	0/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	0/ 0	0/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	0/ 0	0/ 70
2503	0/ 0	0/ 33
2674	0/ 0	0/ 194
2708	0/ 0	0/ 261
2906	0/ 0	0/ 32
307	0/ 0	0/ 35
3335	0/ 0	0/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.7 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	1/ 1	1/ 53
8035	0/ 0	0/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.8: Precision/Recall Details for OSD Tool p0f (SynAck)

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	40/ 40	40/ 52
11763	2/ 6	2/ 2
11820	0/ 0	0/ 70
1331	15/ 15	15/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	46/ 46	46/ 70
2503	0/ 0	0/ 33
2674	84/ 84	84/ 194
2708	0/ 0	0/ 261
2906	24/ 24	24/ 32
307	14/ 14	14/ 35
3335	21/ 28	21/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.8 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	41/ 41	41/ 53
8035	112/ 116	112/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.9: Precision/Recall Details for OSD Tool SinFP (passive)

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	0/ 0	0/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	8/ 8	8/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	24/ 24	24/ 70
2503	0/ 0	0/ 33
2674	6/ 6	6/ 194
2708	0/ 0	0/ 261
2906	0/ 0	0/ 32
307	1/ 1	1/ 35
3335	0/ 0	0/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.9 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	0/ 0	0/ 53
8035	36/ 36	36/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.10: Precision/Recall Details for OSD Tool Siphon

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	7/ 7	7/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	0/ 0	0/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	34/ 34	34/ 70
2503	0/ 0	0/ 33
2674	30/ 30	30/ 194
2708	0/ 0	0/ 261
2906	5/ 5	5/ 32
307	5/ 5	5/ 35
3335	1/ 5	1/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.10 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	7/ 7	7/ 53
8035	20/ 20	20/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Table A.11: Precision/Recall Details for OSD Tool Xprobe

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	48/ 48	48/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	12/ 12	12/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	70/ 70	70/ 70
2503	0/ 0	0/ 33
2674	120/ 120	120/ 194
2708	0/ 0	0/ 261
2906	20/ 20	20/ 32
307	20/ 20	20/ 35
3335	25/ 31	25/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	130/ 130	130/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.11 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	52/ 52	52/ 53
8035	84/ 84	84/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	2/ 2	2/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

A.3 Precision/Recall Details for AppD Tools

Table A.12: Precision/Recall Details for AppD Tool ettercap (passive)

Measure / BID	Precision	Recall
10078	182/ 182	182/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	0/ 0	0/ 52
11763	0/ 0	0/ 2
11820	29/ 29	29/ 70
1331	0/ 0	0/ 15
1578	3/ 3	3/ 10
1806	124/ 124	124/ 221
2124	0/ 0	0/ 172
2417	0/ 0	0/ 70
2503	15/ 15	15/ 33
2674	101/ 101	101/ 194
2708	159/ 159	159/ 261
2906	0/ 0	0/ 32
307	13/ 13	13/ 35
3335	0/ 0	0/ 26
3581	51/ 53	51/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133

Continued on next page

Table A.12 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
529	23/ 23	23/ 28
5556	0/ 0	0/ 12
7106	20/ 20	20/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	0/ 0	0/ 53
8035	0/ 0	0/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	143/ 143	143/ 213

Table A.13: Precision/Recall Details for AppD Tool Nmap

Measure / BID	Precision	Recall
10078	236/ 236	236/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	0/ 0	0/ 52
11763	0/ 0	0/ 2
11820	70/ 70	70/ 70
1331	0/ 0	0/ 15
1578	3/ 3	3/ 10
1806	182/ 182	182/ 221
2124	0/ 0	0/ 172
2417	0/ 0	0/ 70
2503	29/ 29	29/ 33
2674	156/ 156	156/ 194
2708	224/ 224	224/ 261
2906	0/ 0	0/ 32
307	35/ 35	35/ 35
3335	0/ 0	0/ 26
3581	57/ 59	57/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	0/ 0	0/ 133
529	27/ 27	27/ 28

Continued on next page

Table A.13 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	42/ 42	42/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	0/ 0	0/ 53
8035	0/ 0	0/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	177/ 177	177/ 213

A.4 POSD Precision/Recall Details

Table A.14: Precision/Recall Details for POSD

Measure / BID	Precision	Recall
10078	0/ 0	0/ 285
10108	0/ 0	0/ 59
10115	0/ 0	0/ 53
10116	0/ 0	0/ 180
1163	36/ 36	36/ 52
11763	0/ 0	0/ 2
11820	0/ 0	0/ 70
1331	15/ 15	15/ 15
1578	0/ 0	0/ 10
1806	0/ 0	0/ 221
2124	0/ 0	0/ 172
2417	63/ 63	63/ 70
2503	0/ 0	0/ 33
2674	115/ 115	115/ 194
2708	0/ 0	0/ 261
2906	19/ 19	19/ 32
307	20/ 20	20/ 35
3335	11/ 16	11/ 26
3581	0/ 0	0/ 69
4006	0/ 0	0/ 4
4482	0/ 0	0/ 148
4485	0/ 0	0/ 139
5033	0/ 0	0/ 69
514	97/ 97	97/ 133
529	0/ 0	0/ 28

Continued on next page

Table A.14 – Precision/Recall Details - continued from previous page

Measure / BID	Precision	Recall
5556	0/ 0	0/ 12
7106	0/ 0	0/ 262
7116	0/ 0	0/ 120
7230	0/ 0	0/ 2
7294	0/ 0	0/ 386
754	53/ 53	53/ 53
8035	130/ 134	130/ 136
8205	0/ 0	0/ 370
8315	0/ 0	0/ 466
8459	0/ 0	0/ 23
8615	0/ 0	0/ 35
8826	0/ 0	0/ 32
9633	0/ 0	0/ 18
9635	0/ 0	0/ 87
9751	0/ 0	0/ 213

Appendix B

OSD Tests

B.1 Test Descriptions

Definition B.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, well-formed, and exclusively passive. ◯

Definition B.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, well-formed, and can be used both passively and actively[1]*.* ◯

Definition B.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 well-formed, and is exclusively passive. ◯

[1] By forging a SYN packet where the sender IP is an unused address.

Definition B.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 well-formed, and is both active and passive. ◯

Definition B.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 well-formed, and is both active and passive. ◯

Definition B.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 well-formed and can be used both actively and passively. ◯

Definition B.7 (Test-7 (ICMP ID SEQ))

Similar to the IP ID test (see Test-5 in Definition B.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 well-formed, and passive. ◯

Definition B.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 port 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 well-formed and can be active as well as passive. ◯

Definition B.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 B.8). This test is stimulus-response, it is well-formed, and both active and passive. ◯

Definition B.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 well-formed and both active and passive. ◯

Definition B.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 well-formed and can be used both actively and passively. ◯

Definition B.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, it is well-formed/malformed (well-formed because it was once part of a protocol standard, but malformed because it is now obsolete, see Section 4.3.3.7 of RFC 1812) and can be used both actively and passively (although we should not expect to se this kind of messages naturally on the network). ◯

Definition B.13 (Test-13 (ICMP TS))

Exactly like Test-11 (see Definition B.11) except that it relies on the pair ICMP Timestamp request/reply. ◯

Definition B.14 (Test-14 (ICMP Mask))

Exactly like Test-11 (see Definition B.11) except that it relies on the pair ICMP Mask request/reply. ◯

Definition B.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, well-formed, and active only. ◯

Definition B.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 malformed (a TCP packet should always have some flags set), and active only. ◯

Definition B.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 malformed (Syn and Fin flags cannot be set together), and is only active. ◯

Definition B.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, well-formed and active only. ◯

Definition B.19 (Test-24 (Ack closed))
Exactly like Test-23, except that the stimulus is now sent on a closed port. ◯

Definition B.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, well-formed and active only. ◯

Definition B.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 B.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

B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 B.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 C

Operating System Discovery Experiment Information

C.1 Precision Details for Classical OSD Tools

Table C.1: Precision Details for OSD Tool p0f (StrayAck)

Size of Set of Possible OSes	Nb of Occurrence
5	6
7	22

Table C.2: Precision Details for OSD Tool p0f (Syn)

Size of Set of Possible OSes	**Nb of Occurrence**
6	16
41	119

Table C.3: Precision Details for OSD Tool Siphon

Size of Set of Possible OSes	**Nb of Occurrence**
10	161
13	22

Table C.4: Precision Details for OSD Tool SinFP

Size of Set of Possible OSes	**Nb of Occurrence**
1	47
2	48
3	54
4	151
5	122
16	140

Table C.5: Precision Details for OSD Tool p0f (SynAck)

Size of Set of Possible OSes	**Nb of Occurrence**
1	73
2	250
3	60
4	223
12	140

Table C.6: Precision Details for OSD Tool p0f (RstAck)

Size of Set of Possible OSes	**Nb of Occurrence**
1	34
19	1427

Table C.7: Precision Details for OSD Tool nmap

Size of Set of Possible OSes	**Nb of Occurrence**
1	137
2	55
3	242
4	57
6	1043

Table C.8: Precision Details for OSD Tool ettercap

Size of Set of Possible OSes	**Nb of Occurrence**
1	188
2	116
8	42
10	104
11	15
12	1379
13	208
19	140
25	5
27	111
44	986

Table C.9: Precision Details for OSD Tool xprobe

Size of Set of Possible OSes	**Nb of Occurrence**
4	359
8	57
10	817
11	946
13	548
14	163
15	1895
16	148

C.2 POSD Precision Details

Table C.10: Precision Details for POSD

Size of Set of Possible OSes	**Nb of Occurrence**
1	92
2	508
3	228
4	2
5	142
6	324
7	29
9	32
10	8
11	25
12	60
13	795
15	26
17	22
18	79
19	6
24	169
26	6
27	151
29	223
32	1
33	57
35	1
36	598
38	3
40	1
42	1
86	9
190	18

Bibliography

[1] Annie De Montigny-Leboeuf. A Multi-Packet Signature Approach to Passive Operating System Detection. Technical Report CRC-TN-2005-001, Communications Research Center Canada, January 2005.

[2] Ofir Arkin and Fyodor Yarochkin. Xprobe Homepage. `http://xprobe.sourceforge.net`.

[3] Patrice Auffret. SinFP Homepage. `http://www.gomor.org/cgi-bin/sinfp.pl`.

[4] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[5] 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.

[6] Christian Bauer. Basilisk Homepage. `basilisk.cebix.net`.

[7] 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.

[8] R. Braden. RFC 1122 - Requirements for Internet Hosts - Communication Layers. `http://www.faqs.org/rfcs/rfc1122.html`.

[9] Ron Carwell and Heidi Webb. *Guide to Microsoft Virtual PC 2007 and Virtual Server 2005*. Thomson Course Technology, 2007.

[10] Gary Chartrand and Linda Lesniak. *Graphs & Digraphs*. Chapman & Hall, 2004.

[11] François Gagnon. HOSD Homepage. `http://hosd.sourceforge.net`.

[12] François Gagnon. VNEC Homepage. `http://vnec.sourceforge.net`.

[13] François Gagnon. Operating System Discovery Using Answer Set Programming. ACAI 2007 summer school - Poster Session `http://www.sce.carleton.ca/~fgagnon/Publications/PosterACAI2007.pdf`, 2007.

[14] 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.

[15] 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.

[16] François Gagnon and Babak Esfandiari. A Query-Based Approach for Test Selection in Diagnosis - Operating System Discovery as a Case Study. *Submitted to Artificial Intelligence Review Journal - Special Issue on AI & Pervasive Computing*, 2008.

[17] François Gagnon and Babak Esfandiari. Gathering Context for Intrusion Detection - A Study of Operating System Discovery. *Submitted for Publication to IEEE Transactions on Network and Service Management*, 2008.

[18] 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.

[19] 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, School of Computer Engineering - Carleton University, 2008. `http://www.sce.carleton.ca/~fgagnon/Publications/context.pdf`.

[20] Communication Research Center. CRC Homepage. `http://www.crc.ca`.

[21] Luca Console and Pietro Torasso. A Spectrum of Logical Definitions of Model-Based Diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[22] Burak Dayioglu and Attila Ozgit. Use of Passive Network Mapping to Enhance Signature Quality of Misuse Network Intrusion Detection Systems. *Proceedings of the 16th International Symposium on Computer and Information Science (ISCIS'01)*, 2001.

[23] DBAI. DLV Homepage. `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[24] Johan de Kleer. *Readings in Model-Based Diagnosis*, chapter Focusing on Probable Diagnoses, pages 131–137. Morgan Kaufmann, 1992.

[25] Johan de Kleer. *Readings in Model-Based Diagnosis*, chapter Using Crude Probability Estimates to Guide Diagnosis, pages 118–124. Morgan Kaufmann, 1992.

[26] 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.

[27] Johan de Kleer and Brian C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[28] Hervé Debar and Andreas Wespi. Aggregation and Correlation of Intrusion-Detection Alerts. *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID'01) - LNCS*, 2212:85–103, 2001.

[29] Jeff Dike. *User Mode Linux*. Prentice Hall PTR, 2006.

[30] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A.Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, 2004.

[31] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The Diagnosis Frontend of the DLV System. Technical Report DBAI-TR-98-20, Institut für Informationssyteme, Technische Universität Wien, 1998.

[32] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning, II: The DLV^K System. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[33] Foundstone. OS Identification Methods and Coutermeasures. Foundstom Strategic Security white paper, August 2003.

[34] 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.

[35] 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.

[36] Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation — The A-Prolog Perspective. *Artificial Intelligence*, 138(1–2):3–38, 2002.

[37] Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[38] Bernard Golden. *Virtualization For Dummies*. For Dummies, 2007.

[39] 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.

[40] Christopher Kruegel and William Robertson. Alert Verification: Determining the Success of Intrusion Attempts. *Proceedings of the 1st Workshop on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'04)*, 2004.

[41] Annie De Montigny Leboeuf and Frédéric Massicotte. Passive Network Discovery for Real Time Situation Awareness. *Proceedings of the RTO IST Symposium on Adaptive Defence in Unclassified Networks*, April 2004.

[42] Vladimir Lifschitz. Action Languages, Answer Sets, and Planning. *The Logic Programming Paradigm, A 25-Year Perspective*, pages 357–373, 1999.

[43] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunnigham, and Marc A. Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX'00)*, 2:12–26, 2000.

[44] Kuthonuzo Lurua and Shashank Khanvilkar. Virtual Networking with User-Mode Linux. *Linux for you (Networking Expert)*, January 2005.

[45] 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.

[46] John McHugh. Testing Intrusion Detection Systems: A critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluation as Performed by Lincoln Laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, 2000.

[47] 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.

[48] 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.

[49] Benjamin Morin and Hervé Debar. Correlation of Intrusion Symptoms: An Application of Chronicles. *Proceedings of the 6th Symposium on Recent Advances in Intrusion Detection (RAID'03) - LNCS*, 2820:94–112, 2003.

[50] Alberto Ornaghi and Marco Valleri. Ettercap Homepage. `http://ettercap.sourceforge.net`.

[51] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[52] 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.

[53] Patrick Goldsack et al. SmartFrog: Configuration and Automatic Ignition of Distributed Applications. *HP OVUA'03*, 2003.

[54] Samuel Patton, William Yurcik, and David Doss. An Achilles' Heel in Signature-Based IDS: Squealing False Positives in SNORT. *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID'01)*, 2001.

[55] Vern Paxson. Bro: A System for Detecting Intruders in Real-Time. *Proceedings of the 7th USENIX Security Symposium*, 1998.

[56] 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.

[57] David Poole. Representing Knowledge for Logic-Based Diagnosis. *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1282–1290, 1988.

[58] David Poole. Representing Diagnosis Knowledge. *Annals of Mathematics and Artificial Intelligence*, 11(1-4):33–50, March 1994.

[59] 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.

[60] Raymond Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[61] J. Reynolds and J. Postel. RFC 1700 - Assigned Numbers. `http://www.faqs.org/rfcs/rfc1700.html`.

[62] Tenable Netwok Security. Nessus 3.2 Advanced User Guide (Revision 9). `http://www.nessus.org/documentation/nessus_3.2_advanced_user_guide.pdf`, March 2008.

[63] SecurityFocus. SecurityFocus Homepage. `http://www.securityfocus.org/`.

[64] Umesh Shankar and Vern Paxson. Active mapping: Resisting NIDS Evasion Without Altering Traffic. *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 44–61, 2003.

[65] Jerry Shenk and Dave Shackleford. Sourcefire Real-Time Network Awareness. SANS Analyst Program.

[66] Matthew Smart, Robert Malan, and Farnam Jahanian. Defeating TCP/IP Stack Fingerprinting. *Proceedings of the 9th USENIX Security Symposium*, pages 229–240, 2000.

[67] Robin Sommer and Vern Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 262–271, 2003.

[68] Subterrain Security Group. Siphon Homepage. `http://siphon.datanerds.net/`.

[69] Greg Taleck. Ambiguity Resolution via Passive OS Fingerprinting. *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID'03) - LNCS*, 2820:192–206, 2003.

[70] Tenable Network Security. Nessus Homepage. `http://www.nessus.org`.

[71] 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.

[72] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.

[73] Vladimir Lifschitz. Answer Set Planning. *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, 1999.

[74] Vladimir Lifschitz. Introduction to Answer Set Programming. Material for a course at ESSLLI 2004. `http://www.cs.utexas.edu/users/vl/mypapers/esslli.ps`, 2004.

[75] VMWare. VMWare Homepage. `http://www.vmware.com/`.

[76] VMWare. Introducing the VIX API. *VMWorld'06*, 2006.

[77] Steven Warren. *The VMWare Workstation 5 Handbook (Networking and Security)*. Charles River Media, 2005.

[78] Fyodor Yarochkin. Nmap Homepage. `http://www.insecure.org/nmap/`.

[79] Fyodor Yarochkin. Remote OS detection via TCP/IP Stack FingerPrinting. `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`, October 1998.

[80] Michal Zalewski. p0f homepage. `http://lcamtuf.coredump.cx/p0f.shtml`.

[81] Jingmin Zhou, Adam Carlson, and Matt Bishop. Verify Results of Network Intrusion Alerts Using Lightweight Protocol Analysis. *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.