# A Hybrid Approach to Operating System Discovery using Answer Set Programming

François Gagnon
fgagnon@sce.carleton.ca
www.sce.carleton.ca/~fgagnon
Carleton University, Canada

Babak Esfandiari
babak@sce.carleton.ca
www.sce.carleton.ca/faculty/esfandiari
Carleton University, Canada

Leopoldo Bertossi
bertossi@scs.carleton.ca
www.scs.carleton.ca/~bertossi
Carleton University, Canada

***Abstract*—The goal of operating system (OS) discovery is to learn which OS is running on a distant computer. There are two main strategies for OS discovery: active and passive. Each of them has advantages as well as drawbacks. This paper discusses how answer set programming, a new logic programming paradigm, can be used to address, in a simple and elegant way, the problem of operating system discovery in computer networks by logically specifying the problem and providing solutions through automated reasoning. As a result of using such a knowledge representation framework, it is possible to unify the active and the passive methods to OS discovery in a single hybrid approach that has the advantages of both strategies while being much more versatile. Moreover, this paper presents a proof of concept prototype for hybrid operating system discovery.**

## I. 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 to know what are the operating systems (OS) out 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. Recently, researchers in security have begun to look at the possibility of using the OS information of a machine to correlate if an attack attempt reported by an intrusion detection system (IDS) has some chances to succeed on the specified target or not [1], [2]. Furthermore, [3] discusses different strategies to gather OS information in the context of IDS; and it is clear that neither passive nor active techniques are ideal for this purpose. On the other hand, it seems likely that a hybrid approach, such as the one proposed here, would work well. In this paper, we focuss on using operating system discovery (OSD) in the context of intrusion detection.

There are many requirements for OSD in the context of intrusion detection. First, it is not appropriate to generate abnormal network traffic nor huge amount of normal traffic in order to obtain information about a machine's OS (thus current active technique are not suitable). Second, when information on a given machine is needed, this information must be provided as soon as possible (thus a purely passive technique is not ideal). Third, the queries to a OSD tool would be of the form "Does machine $Ip$ run the operating system $O$?", or in a more general way "Is machine $Ip$ running an operating system that belongs to the set $\Theta$?". State of the art OSD tools are not able to answer such queries without going through the, often useless, work of solving the query "Which operating system is running on machine $Ip$?". Finally, it is important to have an easy and flexible way of rapidly updating the rules used to deduce OS, to take care of new operating system releases.

### A. Contributions

In this paper, we make 3 contributions.

- We propose a new strategy for OS discovery called hybrid operating system discovery.
- We argue that hybrid OSD needs an appropriate knowledge representation and formal specification language, and that Answer Set Programming (ASP)[1] [5] is a judicious choice for such a language. Some powerful implementations of ASP already exist, among others *DLV* [6] and *SMODELS* [7].
- To validate our hybrid OSD approach and demonstrate its feasibility, we present a proof of concept tools and some preliminary experiment results.

The hybrid approach proposed in this paper works as follows. When the system receives a query about the OS running on a given machine, it tries to use past events to deduce the OS. When it is not possible, the system will use as few active tests as possible to gather the missing information (instead of using all available tests). This should generate a small amount of network traffic, but still identify the OS very fast (in fact faster than with purely active techniques since only some active tests will be performed). Our hybrid approach is detailed in Section III. As far as we know, this is the first attempt to design a hybrid approach to OS discovery, and certainly the first to use ASP for this purpose.

To allow passive and active OS discovery to coexist in an elegant and useful way, there are some requirements on the underlying language used for knowledge representation. First, a declarative language is preferable since it gives the possibility to automatically generate the program from a repository, i.e. from a database of operating system behaviors, thus allowing to easily update the deduction rules. Second, the language

[1] Sometimes called A-Prolog [4], but ASP goes much beyond Prolog in that it is more expressive and it has a clear declarative semantics.

1-4244-0799-0/07/$25.00 ©2007 IEEE

must support non-monotonic reasoning (that is the ability to draw conclusions which can be invalidated when adding new information), see [8]. This feature is important since whenever we add a new network packet originating from machine *Ip*, we want to tighten as much as possible the set of possible OS for *Ip*. Next, the logic must support both reasoning in dynamic domains (to infer knowledge passively in a changing world, i.e. computers reboot, change their network configuration, etc.) and planning (to obtain the missing information actively). Finally, it is important that the logic has a sound and complete semantic to assure that the conclusions we make are exactly the good ones. We chose ASP as our language for hybrid OS discovery since it fulfills all of the above requirements, gives us some other interesting features, and there is some literature on frameworks to use ASP for knowledge-base management and planning, see [9]. Section II-B presents a short introduction to ASP and argues why it is a good choice for the task of hybrid OS discovery.

To apply our approach, we also present a proof of concept prototype built with Java and answer set programming that gathers knowledge about the operating systems on a network. The goal of the prototype is to show how, and to a greater extend why, ASP is a judicious choice for hybrid OS discovery. For the sake of clarity, we present here a simplified version of the prototype in which we use only 8 operating systems (*Windows* 2000, *Windows XP*, *SunOS*, *MacOS*, *Free BSD* 5.0, and *Linux Red Hat* 5.2, 7.1, and 8.0), without trying to get detailed information about the particular version of the OS, such as the service pack for *Windows* systems or the kernel for *Linux* systems. We mainly focus on answering the question: "Is a given machine *Ip* running a given operating system *O*?".

The rest of the paper is structured as follows: first, Section II provides the background material for both OS discovery and ASP; then, Section III describes the ASP implementation of hybrid OS discovery; Section IV discusses some preliminary experimentation done with the prototype; Section V wraps up the paper with some discussion; and finally, Section VI discusses some potential areas for future work.

## II. Background

This section presents basic material on both OS discovery and ASP. For more information about OSD and ASP, the reader is referred to [10], [11] and [4], [5], respectively.

### A. *Operating System Discovery*

There are many ways to do OSD. We focus on the analysis of the communication behavior of a machine (more specifically the content of such communications) to deduce the underlying operating system. The basic idea is that in some specific situations, there is no standardized way to behave and each OS constructor must implement the behavior of their choice. There are two main approaches for network OS discovery: passive (see Section II-A1) and active (see Section II-A2).

*1) Passive OS Discovery:* In passive OS discovery one is only allowed to listen on 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 1.

*Example 1 (passive test):* When capturing the network traffic, if one sees an ARP request from a machine with IP address *Ip* in which the destination MAC address is set to FF:FF:FF:FF:FF:FF, then one can conclude that *Ip* is running either *SunOS* or *MacOS* prior to version 10. If the field contains random uninitialized data, then one can conclude that *Ip* is running *FreeBSD* 4.6, 4.6.2, 4.7, 4.8 or 5.0. All other OS initialize the field to 00:00:00:00:00:00. □

The main problem of this 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. Usually, less information can be deduced from usual communication than from carefully engineered stimulus-response sequences. It seems likely that passive tools should monitor the network and update their *knowledge base* on a continuous basis; but in fact, they simply gather packets for a specific window (time or number of packets) and then deduce information from the data collected in that window (by matching the data against some possible signature), regardless of any other information that could have been known beforehand. In particular, SinFP and P0f seem to use a window of one packet; they analyze each received packet and output the best operating system matching. This implementation choice can be anoying for someone wanting continuous monitoring of the network and greatly limits the ability of passive tools to detect some network events such as IP spoofing, reboot or the presence of Network Address Translation (NAT) devices (P0f detects NAT devices but it requires a module independent of the fingerprinting part). p0f [12] and siphon [13] are examples of passive OSD tools.

*2) Active OS Discovery:* In active OSD one can directly probe a machine in order to deduce its operating system, depending on the reaction to the synthesized stimuli. For instance, one may send an abnormal packet to see how the target will react. Since the packet is malformed, there should be no standard way to react and different operating systems may respond differently. Another example would be to stimulate the target with a normal packet and analyzing the way the response correlate with the stimulus. Example 2 presents a stimulus-response active test using normal traffic. Another kind of active tests consists in placing the target machine in unusual conditions and monitoring how it behaves. However, putting a machine in extreme conditions often results in disrupting normal activities. [14] describes such a test called *SynFlood Resistance* and also presents other active OSD tests.

*Example 2 (active test with normal traffic):* By sending a Syn packet to a closed port of machine *Ip*, we can get a RstAck packet from *Ip* and correlate the response with the

stimulus. For instance, some version of MacOS will set the DF bit of the RstAck packet to the same value as the DF bit in the corresponding Syn packet, SunOS will set it to Yes, and Windows to No. MacOS prior to 9.1 seems to use the TTL of the Syn packet as the TTL for the RstAck and QNX 6.0 even use the TCP options of the Syn packet as the TCP options in the RstAck. As discussed in [15], this test can also be performed passively since the stimulus could be sent by a third party machine as part of normal communication. □

The main problem with active OS discovery is the large amount of traffic generated in order to discover the OS. For instance, most active tools need to know about one open and one closed port on the target to perform the tests. Many of those tools, such as Nmap, perform a port scan before doing any active test. A port scan by itself can sometimes generate more than a thousand network packets. On the other hand, with hybrid OS discovery, the state of the target ports could be inferred passively. Moreover, since there does not exist a single test such that one can be sure to learn the OS, it is necessary to align 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 achieve the goal. Most active tools don't bother with planning and simply execute all available tests. Furthermore, active tools are designed to answer questions of the form "Which OS is running on a given machine?", as it is the case with passive tools. To answer effectively (i.e. without doing all the work to know the exact OS) a less restrictive question, such as "Is a given machine running an OS among $\theta$?", an active tool would have to use planning to generate a judicious sequence of actions; an option that is not available in today's active tools. Another problem of 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, even if the query is the same. It is thus necessary to run all tests again. This is not acceptable in the context of IDS since we expect the same query to be repeated and we don't want to generate too much traffic. Another major drawback of active tools is the injection of abnormal packets on the network; which becomes a huge problem when those packets interfere 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 abnormal packet.

Nmap [16] and Xprobe [17] are well known active OSD tools. Another effort in OSD comes from *Core Security Technologies* where they use neural networks instead of rule matching. Unfortunately, their product is commercial and thus not much information is available. It seems like their tool is active and closely related to Nmap (it uses the same tests). Thus it should suffer the same problems as most active tools.

*3) Hybrid OS Discovery:* In hybrid OS discovery, the tool should continuously monitor the network to passively gather as much information as possible. The tool should enter in active mode only when needed (when a query that cannot be answered with the available information is made) and should use the information gathered passively to minimize the number of active tests performed. Example 3 discusses a situation when a hybrid tool should go into active mode.

*Example 3 (hybrid OS discovery):* Suppose a user wants to know if machine *Ip* is running "*Windows* 2000 server sp1" but the information gathered passively so far only allow us to deduce that it is running a *Windows* system. Here we will use some active tests in order to answer the query. However, the set of active tests should only include tests that will discriminate between different kinds of *Windows* systems. For instance, it would be useless to execute a test that distinguish between *Linux* and Windows systems since we already know that *Ip* is running a *Windows* system. □

The author of RallahB, see [18], claims that his tool can do hybrid OSD. However, RallahB simply does passive OS discovery and if no information is provided by the passive mode, one active test, and always the same, is launched. There is no interaction at all between the passive and the active module. Thus, this is not hybrid OS discovery in the sense we discuss here. We are not aware of any work that effectively combine passive and active OS discovery.

A hybrid approach offers many advantages over an active and a passive one. First, constantly monitoring the network implies using a good knowledge management system which will offer more possibility than usual passive tools (detecting, and reacting to, some network events). Second, the objective to only execute tests that are necessary implies the use of planning which in turn offers more flexibility as to what kind of queries the system can answer. Finally, by combining active and passive, we will reduce the amount of traffic generated (and also the amount of time required) for OS discovery purpose (compared to active tools) while achieving the required level of precision (which is not always the case with passive tools).

### B. 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:

- Is declarative.
- Supports non-monotonic reasoning.
- Can be used for knowledge management and planning.
- 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). The language must support non-monotonic reasoning (that is the ability to draw conclusions that can be retracted as soon as more information becomes available) since 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 conclusion.

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 several limitation 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 explicitly support non-monotonic reasoning. 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 to ASP and then it will be possible to show why ASP is appropriate for hybrid OS discovery.

### C. Answer Set Programming

ASP consists of declarative programming using extended disjunctive logic programs (EDLPs) with an answer set semantic. EDLPs admit rules of the form

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

where each $L_j$ is a classical literal, i.e. an atom $A$ or its classical negation $\neg A$; and $not$ denotes weak negation. 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. Right here we can see two major extensions from Prolog:

- we can have disjunction in the head of rules. A rule like $Q(a) \vee S(a) \longleftarrow P(a)$. means that if $P(a)$ is true, then at least one of $Q(a)$ or $S(a)$ must also be true;
- we have both classical (or strong) negation and weak negation (or negation as failure). $\neg P(a)$ is true iff $P(a)$ is false, while $not\ P(a)$ is true if $P(a)$ is false or the truth value of $P(a)$ is unknown.

These extensions give more expressiveness to ASP. For instance, a rule describing that a TCP Syn packet with the DF bit set and a TTL of 128 must originate from a machine running *Windows* 2000 or *Windows XP* would look like:

$$os(Ip, 2k) \vee os(Ip, xp) \longleftarrow tcp(Ip, _, _, _, yes, syn, 128).$$

Having strong and weak negation allows us to write rules representing laws such as the closed world assumption, something is false unless it is explicitly designed to be true: $\neg os(Ip, OS) \longleftarrow not\ os(Ip, OS)$.

$$P(x) \vee \neg Q(x) \longleftarrow R(x), \neg S(y), not\ T(x), not\ \neg U(x) \quad (1)$$

The rule of Equation 1 is valid and it 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 either $P(x)$ is true or $Q(x)$ is false. We may consider that the program is *ground*, i.e. it has all its variables instantiated in the underlying domain of

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

TABLE I
THE GROUND PROGRAM OF RULE (1)

| | | |
|---|---|---|
| $R(a)$ | | |
| $P(b)$ | | |
| $P(x) \vee Q(x)$ | $\longleftarrow$ | $R(x)$ |

TABLE II
A SIMPLE PROGRAM

the program (its Herbrand universe, see [5]). For instance, the ground program formed by the rule of Equation (1) together with the facts $R(a)$ and $S(b)$ (where $a$ and $b$ are constants) is shown in table I.

*Definition 1 (Answer Set):* An answer set $S$ of a program $\Pi$ is some particular set of ground literals such that:

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

Note that a program may have zero, one or more answer sets. If an answer set contain both a literal and its negation ($L$ and $\neg L$), then the program is considered to be inconsistent.

*Example 4 (answer sets):* Let's consider the program of table II. 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 since 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 since it does not satisfy the constraint induced by the rule $P(x) \vee Q(x) \longleftarrow R(x)$ (since $R(a)$ is true, one of $P(a)$ or $Q(a)$ must also be true). □

ASP can be queried to know if a ground literal $L$ is true in a given program $\Pi$. The querying process of ASP offers two different reasoning modes. The **cautious** reasoning claims $L$ is true when $L$ appears in all the answer sets of $\Pi$. The **brave** reasoning sanctions $L$ as being true when $L$ appears in at least one of $\Pi$'s answer sets. By combining the two reasoning modes, we can get a three-valued logic(*true*, *false*, *unknown*).

As we can see, ASP is a declarative programming language, but unlike Prolog, which uses top-down and depth-first, the evaluation strategy is not fixed in ASP and the semantics assure 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 some meta-rules, like hard, weak and weighted 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, then the models violating as few weak constraints

as possible are returned. Weighted constraints are essentially the same thing but a cost is associated to every weak constraint and instead of minimizing the total number of violated weak constraints, the goal is to minimize the total cost of the violated weak constraints.

Answer set programming can be used to represent knowledge, including non-monotonicity, and reason on the basis of this knowledge in a non-trivial way. The domain can be specified in a declarative manner and problem solving becomes a reasoning task. For example, many problems of a combinatorial nature can be represented and solved under this powerful and expressive programming paradigm [5]. ASP has also been used for planning purposes [19], [20], [21], [22]. Here, one specifies the initial situation, the goal situation, and a description of the possible actions together with their effects. The system should return a series of actions that will lead from the current situation to the goal situation. Finally, as discussed earlier, ASP is more expressive than Prolog, allowing disjunction in the head of rules and strong negation. These two extensions will allow to write deductive rules in a simple and intuitive way. The interested reader is referred to [5] for a deeper presentation of ASP.

## III. OS Discovery with ASP

In this section we show how to combine knowledge base management (passive module) and planning (active module) to do both passive and active OSD using ASP. The hybrid OSD tool we present works as follows. First, the passive module gathers network packets and update the knowledge base. When a user makes a query to the system to know if a given machine *Ip* runs a given OS $O$, the system consults its knowledge base to get the set of possible OS $P$ for *Ip*. If the knowledge base contains enough information to answer the query, the answer is returned (yes when $P = \{O\}$, no when $O \notin P$). Otherwise, planning is used to generate a series of active tests to reduce $P$ enough so that the query can be answered. The rest of the section is structured as follows: Section III-A explains how to perform passive OSD using ASP and Section III-B presents active OS discovery.

### A. Passive OS discovery with ASP

The passive OS discovery module is essentially a knowledge base; it is updated for every new captured network packet and queried whenever one wants to know the operating system of a given machine. Section III-A1 presents the Intensional DataBase (IDB) file containing ASP rules that model the behavior of different operating systems; Section III-A2 presents the Extensional DataBase (EDB) file containing the recorded network packets and explains the querying process and Section III-A3 discusses the kind of queries that is supported by the passive module vs other passive tools.

*1) Passive IDB:* Figure 1 presents a fragment of our actual IDB file for passive OS discovery (*passiveOSfingerprinting.IDB*). The first rule is a weak constraint stating that unless it is 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 OS for the computers).

```
%One IP should not correspond to two different OS
:~ os(X,Y), os(X,Z), Z != Y.
%ARP Request
os(X,windows2000) ∨ os(X,windowsXP) ∨
   os(X,linuxRedHat7_1) ∨ os(X,linuxRedHat5_2) ∨
   os(X,linuxRedHat8_0) :- 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,linuxRedHat5_2) :- tcp(X,_,_,_,no,syn,64).
os(X,windows2000) ∨ os(X,windowsXP) :- tcp(X,_,_,_,yes,syn,128).
os(X,freeBSD5_0) ∨ os(X,linuxRedHat7_1) ∨ os(X,linuxRedHat8_0) ∨
   os(X,macOS) ∨ os(X,sunOS) :- tcp(X,_,_,_,yes,syn,64).
%TCP SynAck
os(X,linuxRedHat5_2) :- tcp(Y,X,Yport,Xport,_,syn,_),
   tcp(X,Y,Xport,Yport,no,syn_ack,64).
os(X,windows2000) ∨ os(X,windowsXP) :- 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,linuxRedHat7_1) ∨ os(X,linuxReHat8_0) :-
   tcp(Y,X,Yport,Xport,_,syn,_), tcp(X,Y,Xport,Yport,yes,syn_ack,64).
```

Fig. 1. Some Rules for Passive OS discovery

The set of rules in the group **ARP Request** models different behaviors of an OS regarding the destination MAC address in an ARP request (as explained in Example 1). 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, represent 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.

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. Here again the value of the TTL as well as the DF bit are monitored.

*2) Passive EDB and Querying:* Everytime 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*). It is possible to get an overall view of the knowledge we have so far concerning which OS is possibly running on every computer. To do this, we simply invoke *DLV* asking for every possible answer set. Below are two examples to explain how passive OS discovery works in ASP.

*Example 5 (a first packet):* Suppose the only captured packet 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 $arp(ip10_1_1_6, ip10_1_1_1, 1, mac00_00_00_00_00_00)$. By asking *DLV* to compute all possible answer sets, we end up with five of them. Each answer set contains one possible operating system for 10.1.1.6. The first rule of the **ARP Request** group (see Figure 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

either *Windows* 2000, *Windows XP*, *Linux RedHat* 5.2, *Linux RedHat* 7.0 or *Linux RedHat* 8.0. □

*Example 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 RedHat* 7.0 and *Linux RedHat* 8.0. The answer corresponds to the intersection of the possible OS for each rule triggered by the facts (the first rule of the **ARP Request** group and the last rule of the **TCP Syn** group, see Figure 1). □

At the end of Example 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 it is not allowed thanks to the weak constraint discussed at the beginning of Section III-A1 that prevents a machine to be assigned multiple OS when a single one is sufficient. However, there exist some situations where the knowledge base will contain some facts such that 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 are behind a Network Address Translation (NAT) module, they may generate packets containing different OS fingerprint all generated from the same IP address. The weak constraint is flexible enough to permit such situations while making a difference between OS ambiguity and multiple OS hidden behind a single IP address.

One drawback of answer sets is the combinatorial explosion. If we put packets for 2 different IP addresses in the EDB file, each answer set will contain a possible OS for each machine. The number of answer sets can thus become very large. Example 7 present an example of such combinatorial explosion. To circumvent this problem, we can maintain one EDB file for each IP address. This is discussed in Section IV.

*Example 7 (a third packet):* At the end of Example 6, we know that 10.1.1.6 is running either *Linux RedHat* 7.0 or *Linux RedHat* 8.0. Suppose we had a third packet which is an ARP request from 10.1.1.7 to 10.1.1.1 with the destination MAC address set to all zeroes ($arp(ip10_1_1_7, ip10_1_1_1, 1, mac00_00_00_00_00_0)$). A quick look at Figure 1 allows us to deduce that 10.1.1.7 is running either *Windows* 2000, *Windows XP*, *Linux RedHat* 5.2, *Linux RedHat* 7.0 or *Linux RedHat* 8.0. Since there is two possible OS for 10.1.1.6 and 5 for 10.1.1.7, there will be 10 answer sets (one to encode each possible assignment). By splitting the EDB file in two (the packets that belongs to 10.1.1.6 vs those that belongs to 10.1.1.7), we would have only 7 answer sets (2 for 10.1.1.6 and 5 for 10.1.1.7). Such an optimization becomes increasingly important as the number of monitored hosts grows. □

*3) Queries for the Passive Module:* While other passive fingerprinting tools are designed to output the exact operating system (when such information is available) running on a machine, the one presented in this section can provide a set of possible operating systems for a machine. To see the benefits of having the set of possible OS instead of the exact OS, we present how two different systems would answer 3 different queries.

"Is machine *Ip* running the OS $O$?"

- $O'$ is the exact OS: if $O'$ is defined, then the answer would be *yes* if $O = O'$ and *no* otherwise ($O \neq O'$). Whenever $O'$ is not defined, the answer would be *maybe*.
- $P$ is the set of possible OS: the answer would be *yes* if $P = \{O\}$, *no* if $O \notin P$ and *maybe* otherwise.

"Is machine *Ip* running an OS $\in \theta$?"

- $O'$ is the exact OS: if $O'$ is defined, then the answer would be *yes* if $O' \in \theta$ and *no* otherwise ($O' \notin \theta$). Whenever $O'$ is not defined, the answer would be *maybe*.
- $P$ is the set of possible OS: the answer would be *yes* if $P \subseteq \theta$ *no* if $P \cap \theta = \emptyset$ and *maybe* otherwise.

"Which OS is running on *Ip*?"

- $O'$ is the exact OS: if $O'$ is defined, then the answer would be $O'$. Otherwise, the answer would be *unknown*.
- $P$ is the set of possible OS: if $P = \{O\}$, then the answer would be $O$. Otherwise, the answer would be *unknown*.

Note that since most passive tools only give an answer when they know the exact OS, they won't be of any help to answer these queries unless they know the exact OS. On the other hand, our tool will sometimes be able to answer those queries without knowing the exact OS.

### B. Active OS Discovery with ASP

When a query is made to the knowledge base to know if a given machine *Ip* is running a given operating system $O$ and the query result turns out to be "unknown", then it would be interesting to build a plan (a series of active OS discovery tests) such that after the execution of those tests, the knowledge base will have enough information to decide if *Ip* is running $O$. So here, we use the planning abilities of ASP. Section III-B1 presents the IDB file containing the ASP description of possible actions (the active OS discovery tests) and Section III-B2 describes the EDB file containing the description of the initial situation and explains how to query the system for a plan. Finally, Section III-B3 discusses the kind of queries that are supported by the active module vs other active tools while Section III-B4 discusses how we can build "good" plans using DLV as the engine to compute the answer sets.

*1) Active IDB:* A fragment of our actual intensional database (*activeOSdiscovery.IDB*) is presented in Figure 2. The first rule states that tests are executed only if necessary (as few tests as possible are used in a plan). This rule is very important in our planning module and cannot be used in less expressive languages, like Prolog, that do not support weak constraints. The next four rules[2] define the planning, their meaning is:

[2]The predicate *holds/3* expresses the possible OS at each state while *possible/3* denotes what is possible with respect to the outcome of 1 test.

1) if an association $\langle IP, OS \rangle$ does not hold before the execution of an action, it will not hold after;
2) if an association $\langle IP, OS \rangle$ holds before the execution of an action and the outcome of this action confirm that possibility, then $\langle IP, OS \rangle$ still holds after the execution of the action;
3) all association $\langle IP, OS \rangle$ that are true remain true when no action is executed;
4) what is not explicitly said to be possible by the outcome of a test should be considered as not being possible.

Again here, some of those rules would not be expressible in Prolog since they use strong negation. The last part of Figure 2 describes one possible active test, namely the **TCP SynAck**. Note that this active test corresponds to the passive knowledge updating rules presented in the **TCP SynAck** group of Figure 1. If the TCP SynAck test is executed against *Ip* at time $T$, this will cause at least (and exactly) one of the predicates $g1A(Ip,T1)$, $g1B(Ip,T1)$, $g1C(Ip,T1)$, or $g1D(Ip,T1)$ to be true. When such a predicate is true, this means that only a certain set of OS is possible for the given *Ip*. Thus here the set of possible OS is the intersection of what was possible before the execution of the test with what is possible with respect to the result of the test. Which of these $g1$ predicates will end up being true depends on the actual result of the test and thus on the actual operating system running on the machine. Example 8 details the effects of an action. A complete example of planning will be presented in Section III-B2.

*Example 8 (effects of actions):* Suppose that the test TCP SynAck is executed against *Ip* at time $T$ where we know that *Ip* is running Linux Red Hat, but we don't know if it is running 5.2, 7.1, or 8.0. Then at time $T1$, exactly one of $g1A(Ip,T1)$, $g1B(Ip,T1)$, $g1C(Ip,T1)$, or $g1D(Ip,T1)$ will be true. Now suppose the result of the test is such that $g1D(Ip,T1)$ is true. With the definition of **g1D** from Figure 2 we can see that $possible(IP, freeBSD5_0, T1)$, $possible(IP, linuxRedHat7_1, T1)$ and finally $possible$ $(IP, linuxRedHat8_0, T1)$ are forced to be true. Thus we can conclude that *Ip* is running *Linux Red Hat* 7.1 or 8.0. *Ip* cannot be running *Linux Red Hat* 5.2 since the test result eliminates it (*possible/3* is not true in rule 2) and cannot be running *Free BSD* 5.0 since it was not a possibility before the execution of the test (*holds/3* is not true in rule 2). □

*2) Active EDB and Querying:* What is interesting in using both active and passive OS discovery techniques in a tool is that the knowledge acquired by the passive module can serve to reduce the amount of work that has to be done actively, see Example 9.

*Example 9 (using passive and active information):* Suppose the current knowledge allows us to state that *Ip* is running either *Linux Red Hat* 7.1 or *MacOS*. Then it is possible to learn the actual operating system by doing a single test, namely the TCP SynAck test. On the other hand, if we do not use the information provided by the knowledge base and assume that *Ip* can be running any operating system, it may not be possible to learn the exact OS of *Ip* by doing only the TCP SynAck test. We may have to do more tests. □

```
%Execute actions only when needed
:~ execute(_,_,_).
%Planning
1) -holds(IP,OS,T1) :- -holds(IP,OS,T), T < T1.
2) holds(IP,OS,T1) :- holds(IP,OS,T), possible(IP,OS,T1), next(T,T1).
3) holds(IP,OS,T1) :- holds(IP,OS,T), not actionExecuted(IP,T), next(T,T1).
4) -possible(IP,OS,T) :- not possible(IP,OS,T).
%The TCP SynAck test
g1A(Ip,T1) v g1B(Ip,T1) v g1C(Ip,T1) v g1D(Ip,T1) :-
    next(T,T1), execute(testTCP_SynAck,Ip,T).

%g1A (os is linux Red Hat5.2)
possible(IP,linuxRedHat5_2,T1):- g1A(IP,T1).
%g1B (os is windows 2000 or XP)
possible(IP,windows2000,T1):- g1B(IP,T1).
possible(IP,windowsXP,T1):- g1B(IP,T1).
%g1C (os is mac or sun OS)
possible(IP,macOS,T1):- g1C(IP,T1).
possible(IP,sunOS,T1):- g1C(IP,T1).
%g1D (os is free BSD 5.0 or linux Red Hat 7.1 or 8.0)
possible(IP,freeBSD5_0,T1):- g1D(IP,T1).
possible(IP,linuxRedHat7_1,T1):- g1D(IP,T1).
possible(IP,linuxRedHat8_0,T1):- g1D(IP,T1).
```

Fig. 2. Some Rules for Active OS Discovery

Passively gathered information can be used to reduce the amount of active work in the following way. To learn if a given operating system $O$ is running on machine *Ip*, the first thing to do is to ask for an overview of the knowledge concerning machine *Ip*; we get $P$, the set of currently possible OS for *Ip*. If $O \notin P$, then clearly *Ip* is not running $O$. If $P = \{O\}$, then clearly *Ip* is running $O$. Otherwise, some active tests have to be performed in order to discover if *Ip* is running $O$. To know which tests should be performed, planning can (and should) be used. The current state needs to be given to the planning module together with a goal state. This information will be written in the EDB file (*activeOSDiscovery.EDB*). Example 10 explains the process of planning using the current knowledge.

*Example 10 (planning actions):* Suppose that the current knowledge base allows us to infer that machine 10.1.1.5 is running either *Windows* 2000, *SunOS*, or *Linux Red Hat* 8.0 and that we want to know if 10.1.1.5 is running *Windows* 2000. Since it is possible but not certain that 10.1.1.5 is running *Windows* 2000, we must rely on active OS discovery through planning. Figure 3 presents the content of our EDB file for this particular case. The first group with predicate $holds(ip10_1_1_5, p, 0)$, where $p$ represents the three possible OS, describes the initial state. The next line imposes that there is only one possible OS for 10.1.1.5 at the goal state[3]. The last line corresponds to the query about the possibility to reach the given goal state. If we ask *DLV* to solve this planning problem it will give us a plan to execute the TCP SynAck test. Doing this test will result in updating the knowledge base with the result of the test and from there the same query to the knowledge base will return either yes or no depending if 10.1.1.5 is actually running *Windows* 2000 (the outcome of the test is $g1B$) or not (the outcome of the test is anything but $g1B$). □

[3] Here we make the assumption that it is always possible to distinguish between two different OS (otherwise it would not always be possible to find a plan). One way to make such a assumption true would be to group all OS that behave exactly the same under a unique label and use this label in the deduction rules and querying.

```
holds(ip10_1_1_5, windows2000,0).
holds(ip10_1_1_5, sunOS,0).
holds(ip10_1_1_5, linuxRedHat8_0,0).
:- #count{A : holds(ip10_1_1_5,A,4)} > 1.
holds(ip10_1_1_5, windows2000, 4)?
```

Fig. 3. Description of Current State for Planning

*3) Queries for the Active Module:* State of the art active tools for OS discovery aim to learn which operating system is running on a given machine. Answering other questions such as "Is machine *IP* running the OS $O$?" or "Is machine *IP* running an OS that belongs to $\theta$?" can only be done by first learning the exact OS running on machine *IP* and interpreting the result. Here, we discuss how our active module could answer those different queries and we argue that it should require less active tests than other active tools. Note however that the current version of the prototype can only answer queries of the form "Is machine *IP* running the OS $O$?".

The case for the query "Is machine *Ip* running the OS $O$?" has been discussed throughout this section and we already mentioned that the resulting plan will contain as few tests as possible (remember the weak constraint used at the top of Figure 2). Even in the worst case where no information has been gathered passively before the query (the set of possible OS is the set of all OS), we should be able to get the answer without executing all tests since we know the objective $O$.

The query "Is machine *IP* running an OS that belongs to $\theta$?" consists of reducing the set of possible OS $P$ such that either $P \subseteq \theta$ or $P \cap \theta = \emptyset$. The best strategy (remove from $P$ the element that are not in $\theta$ or remove from $P$ the element that are in $\theta$) depends on the number of element of $\theta$ that are in $P$ and the number of element that are in $P$ but not in $\theta$, but it also depends on the actual OS running on *IP*. Thus, even if $P = \theta \cup \{O\}$ it will be essential to bring $P$ to $\{O\}$ if $O$ is the actual OS of *IP*. However, by executing tests that remove as much elements from $P$ as possible, we should reduce the number of tests required far below usual active tools.

The query "Which OS is running on machine *IP*" consists of removing as many elements as possible from $P$ until it becomes a singleton. By executing tests that will partition $P$ into classes as small as possible and since we use passively gathered information to reduce the set of possible OS, we should reduce the number of tests required below what is required by other active tools.

For now, a plan simply consists of a set of actions that can be executed in any order and even all at the same time. But, by using a branching plan, it's possible to generate a nearly optimal plan. A branching plan is a plan where the action to take at a specific time point depends on the outcome of the previous action (see [23]). It has a tree like structure where the root is the first action to take; the level one nodes dictate what is the second action to take depending on the outcome of the first action, and so on. Each leaf contains a final answer.

*4) Building Good Plans:* The main reason why it is important to use planning in order to select the series of actions to be executed is that we want to use the *best* plan available. As discussed earlier, the notion of best include a minimal number of active tests to be performed. Other than that, we can argue that some tests are more costly than other (some tests need more network resources, some are more intrusive than others, some require the use of abnormal network traffic, ...). *DLV* offers a convenient way to assign a cost to each test in such a way that the best plan will correspond to the plan minimizing the total cost. This can be done using weighted weak constraints [6]. If the best plan available is very expensive, one option could be to avoid the execution of such a plan at the cost of being unable to answer the query.

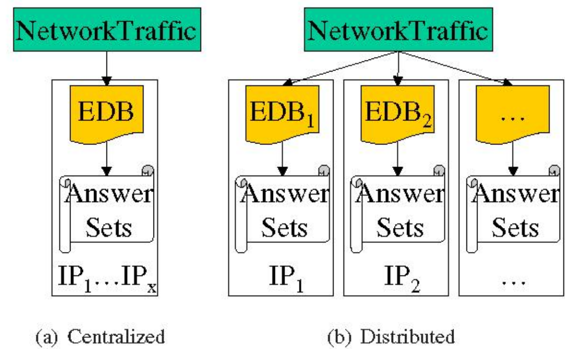


Fig. 4. Operation Modes

## IV. Experiments

### A. Time Benchmarks

Our main concern with ASP was the time complexity of computing the answer sets of a program. To assure that it is a practical solution, we used two different operation modes, distributed and centralized, to evaluate the time required by the passive module[4]. The centralized case consists of placing all recorded packets in a single EDB file, see Figure 4(a); to get a general overview of the knowledge base, one simply needs to compute the answer sets of this extensional database. Here an answer set contains a possible OS for each IP address found in the EDB file. On the other hand, the distributed mode consists of having one EDB file per IP address, see Figure 4(b); to get a general overview of the knowledge base, one must compute the answer sets of each EDB file[5]. Note that the two operation modes are equivalent with respect to knowledge; that is, they both provide the same amount of information from a given set of packets. Thus, it is strictly an implementation choice. Note also that both operation modes could be implemented on a single computer, it is not necessary that the distributed mode be actually distributed over several computers; distribution is over several EDB files (several knowledge bases).

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[6].

[4]The passive module, continuously triggered by the new packets arriving on the network, will be under more stress than the active module, triggered by a user request.

[5]Here, it is possible to get an overview of the knowledge base with respect to a given host without computing the general overview.

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

In centralized mode, we simply executed *DLV* to compute the answer sets and timed the execution. For the distributed mode, 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 of the whole database. The tests were done 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 III.

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 way smaller in the distributed mode, never more than 40 compared to sometimes more than 2000; the minimum number of answer sets is a bit smaller in the centralized mode. The first observation is explained by the combinatorial explosion experienced in the centralized mode. For instance, suppose that at some state of the knowledge base each of the seven hosts has 6 possible OS. In distributed mode, this would require $7 \times 6 = 42$ answer sets, while $6^7 = 279936$ are required in centralized mode. The second observation, while surprising at first, can be explained by recalling that in the centralized mode we compute the answer sets for one EDB file (in the best case this could give one answer set); while for the distributed mode 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. In the centralized case, 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 have only a few packets. For the distributed mode, the time is a lot more stable (always between 0.3 and 0.4 sec). Remember that the number of answer sets produced was also a lot more stable in the distributed mode (between 8 and 38). It is interesting to see the small amount of time required by the distributed mode (0.4 second is always sufficient[7]) and that the number of packets seems to have little influence on the distributed mode.

Thus, using a strategy similar to the distributed mode to avoid combinatorial explosion seems to enable the use of ASP as a practical tool for hybrid OS discovery. Note also that the above analysis is a bit pessimistic. Each run was done starting with an empty knowledge base, where all OS are possible for each machine. We believe that when the knowledge base already contains information, i.e. when the set of possible OS is not the set of all OS, the number of answer sets and thus the time required to compute them will be significantly reduced.

Other preliminary experiments (not shown here) have been done to see the effect of the number of deduction rules and the number of possible OS on the execution time. Not surprisingly, it seems like the number of rules has an impact on the computation time (the bigger is the program, the longer it takes to execute), but the time variation does not seem to grow rapidly. A deeper analysis and more experiments are needed to assure the practicality of ASP on a larger scale.

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

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

TABLE III
TIME BENCHMARKS

### *B. Accuracy*

To test the accuracy of our hybrid tool[8], we set up an experiment to compare our results with those of p0f 2.0.8. For this experiment we used 2949 traffic traces generated by 52 different operating systems. Those traces are the result of attacking one of the 52 targets using a vulnerability exploitation program. Details about those traces can be found in [24].

We choose to use the Syn mode of p0f since it is currently the only non-experimental mode available. Since p0f outputs an answer for each packet, we gathered the set of answers concerning the target of a trace to form the set of possible OS. For 2528 traces, p0f gave no answers for the target. For 16 traces, p0f gave some answers but none were the actual target OS. For only 166 traces did p0f give the actual OS among some wrong answers (usually 30 wrong answers plus the good one); in only 4 cases of those 166 we had a set of possible OS of size less or equal to 10.

The results of our hybird tool look very promising. For only 435 traces out of 2949, the tool gave no answers or too many answers to be useful (more than 140 possibilities out of the initial 171 OS). For 7 traces, the hybrid tool gave some answers but none were the actual target OS. For the 2507 remaining traces, the tool gave the actual OS among some wrong answer (always less than 30 wrong answers plus the good one); in 818 cases out of those 2507, we had a set of possible OS of size less or equal to 10.

The accuracy results presented here seem to support two of our initial assumptions:

- Using a knowledge base to keep previously deduced information enhance the accuracy.
- A fully passive tool may not be sufficient in the context of intrusion detection.

## V. DISCUSSION

In this paper, we presented a hybrid approach to operating systems discovery. The hybrid approach combines the advantages of both passive and active techniques while being much more versatile. The passive module continuously monitors the network allowing to detect abnormal behavior (IP spoofing,

[8]Actually we have tested only the accuracy of the passive module.

NAT, etc.) and is multi-packets based thus allowing a more precise and accurate identification of OS. The active module relies on the information gathered passively to reduce the amount of work needed to answer a query and uses planning the execute only pertinent tests.

We argued that to support such a hybrid approach, the underlying representation language must respect some criteria such as: support non-monotonic reasoning, planning and knowledge management, be declarative and have a sound and complete semantic. As a consequence of using a good knowledge representation language, a hybrid OSD tool can deal with more types of queries than other OSD tools.

Finally, we proposed to use ASP as a knowledge representation language to implement the concept of hybrid OS discovery. ASP seems a judicious choice since it respects all the aforementioned requirements and offers some other interesting features such as: weak and weighted constraints, disjunction in the head of rules and both strong and weak negation. Of course this expressiveness comes at the cost of time complexity. To make sure that ASP is practically suitable for hybrid OS discovery, we build a proof of concept implementation and we presented some experimentation results. Among other things, we discussed a strategy to easily reduce the combinatorial explosion underlying the computation of answer sets in ASP.

## VI. Future Work

Many aspects of the work presented here will be further developed in order to exploit the full power of ASP and to build a tool that is suitable for OS discovery in a real world environment and that could be used by third party applications.

One of the main advantage of using a declarative language such as ASP is the possibility to automatically generate the rule set required by the program. In our particular case, it would be extremely interesting to have a script that automatically generates both the *passiveOSdiscovery.IDB* and the *activeOSdiscovery.IDB* files from a database of OS fingerprints. With such a script, one could update the fingerprint database with new data as soon as a new OS is released (using a tool such as [15]) and then update the intensional database for both the passive and active part of OS discovery.

In order to extend the work presented here, it would be interesting to allow more general queries for the active module. Instead of building a plan to find out whether machine *Ip* runs a given operating system $O$, it would be nice to build a plan to learn if *Ip* runs one of the OS among a given set $\Theta$. Another important query is of course "which OS is running on *Ip*".

Section III-B4 discussed about the possibility to specify a cost to each action. We have not yet fully investigated how this could be applied to the field of OS discovery and we believe there is still interesting features to seek there.

Extending the hybrid approach to application discovery would be extremely relevant from a security point of view.

## Acknowledgement

We would like to thank *Frédéric Massicotte* and *Annie De Montigny-Leboeuf* from Canada's Communication Research Center for their help throughout this project.

## References

[1] F. Massicotte, M. Couture, Y. Labiche, and L. Briand, "Context-Based Intrusion Detection Using Snort, Nessus and Bugtraq Databases," *Proceedings of the Third Annual Conference on Privacy, Security and Trust (PST'05)*, October 2005.

[2] G. Taleck, "Ambiguity Resolution via Passive OS Fingerprinting," *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID'03) - LNCS*, vol. 2820, pp. 192–206, 2003.

[3] R. Gula, "Correlating IDS Alerts with Vulnerability Information," http://www.tenablesecurity.com/images/pdfs/va-ids.pdf, December 2002.

[4] M. Gelfond and N. Leone, "Logic Programming and Knowledge Representation: The A-Prolog Perspective," *Artificial Intelligence*, vol. 138, no. 1-2, pp. 3–38, 2002.

[5] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[6] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV System for Knowledge Representation and Reasoning," *ACM Transactions on Computational Logic (to appear)*, 2007.

[7] T. Syrjänen and I. Niemelä, "The Smodels System," in *Proc. 6th International Conference on Logic Programming and Nonmonotonic Reasoning, (LPNMR 2001)*. Springer LNCS 2173, 2001, pp. 434–438.

[8] W. Lukaszewicz, *Non-Monotonic Reasoning: Formalization of Commonsense Reasoning*, ser. Ellis Horwood Series in Artificial Intelligence, J. Campbell, Ed. New York, Horwood, 1990.

[9] C. Baral and M. Gelfond, "Reasoning Agents in Dynamic Domains," *Proceedings of the 1999 Workshop on Logic-Based Artificial Intelligence (LBAI'99)*, pp. 257–279, 1999.

[10] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein, "Passive Operating System Identification From TCP/IP Packet Headers," *Proceedings of the 2003 Workshop on Data Mining for Computer Security (DMSEC'03)*, 2003.

[11] C. Trowbridge, "An Overview of Remote Operating System Fingerprinting," *SANS InfoSec Reading Room - Penetration Testing*, October 2003.

[12] M. Zalewski, "p0f 2.0.7," http://lcamtuf.coredump.cx/p0f.shtml, August 2006, (August 2006).

[13] Subterrain Security Group, "The Siphon Project," http://siphon.datanerds.net/, 2000, (June 2006).

[14] F. Yarochkin, "Remote OS detection via TCP/IP Stack FingerPrinting," http://www.insecure.org/nmap/nmap-fingerprinting-article.html, October 1998.

[15] A. De Montigny-Leboeuf, "A Multi-Packet Signature Approach to Passive Operating System Detection," Communications Research Center Canada, Tech. Rep. CRC-TN-2005-001, January 2005.

[16] F. Yarochkin, "Nmap," http://www.insecure.org/nmap/, June 2006, (June 2006).

[17] Sys-Security Group, "Xprobe 2.0.3," http://www.sys-security.com/index.php?page=xprobe, July 2005, (June 2006).

[18] C. Wilks, "RallahB A Network Intrusion Detection System (NIDS) With Hybrid Fingerprinting," http://zoo.cs.yale.edu/classes/cs490/00-01b/wilks.cherrielyn.csw9/, April 2001, (June 2006).

[19] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, "A Logic Programming Approach to Knowledge-State Planning, II: The DLV$^K$ System," *Artificial Intelligence*, vol. 144, no. 1-2, pp. 157–211, 2003.

[20] 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*, vol. 5, no. 2, pp. 206–263, 2004.

[21] V. Lifschitz, "Action Languages, Answer Sets, and Planning," *The Logic Programming Paradigm, A 25-Year Perspective*, pp. 357–373, 1999.

[22] V. Lifschitz, "Answer Set Planning," *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pp. 23–37, 1999.

[23] H. Levesque, "What is planning in the presence of sensing?" *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pp. 1139–1146, 1996.

[24] F. Massicotte, F. Gagnon, M. Couture, Y. Labiche, and L. Briand, "Automatic Evalaution of Intrusion Detection Systems," *Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC'06)*, 2006.