Operating System Discovery Using Answer Set Programming

François Gagnon, Ph.D Student, Carleton University
fgagnon@sce.carleton.ca

The goal of operating system (OS) discovery is to learn which OS is running on a remote computer by looking at differences in the TCP/IP stack implementation of different vendors. There are two main strategies for OS discovery: passive, where deductions are made by looking at regular communications between computers, and active, where stimuli are sent to the target to see how it reacts in specific (often non-standard) situations. Each technique has its advantages as well as its drawbacks. The work described here studies how logic programming under the answer set semantics can be used to address, in a simple and elegant way, the task of operating system discovery 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 passive methods for OS discovery in a single hybrid approach that has the advantages of both strategies while being much more versatile.

Current passive tools for OS discovery (OSD) have huge limitations. First, each packet is processed individually, meaning a stimulus-response correlation is not possible. Secondly, they are memoryless; that is, each packet is considered as being the only available information without considering the previous deductions. This greatly limits their accuracy.

While active OSD tools are much more accurate, they also have shortcomings. First, they are usually very noisy (sometimes generating several hundreds of packets to discover the OS of a single host). Secondly, they often generate abnormal traffic (to see how the host reacts in non-standard situations) which may interfere with network monitoring tools such as intrusion detection systems.

To circumvent those problems, we propose to use logic programming to implement a passive OSD module, and planning (on top of the passive module) to implement an active module, in a hybrid approach.

1. Passive Module

The rules forming the passive module will have the following 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 (in predicate logic) or its classical negation $\neg A$; and *not* denotes weak negation. The logic programs are evaluated using the answer set semantics.

Such rules offer an intuitive way to specify the deduction engine of the passive module for OSD. For instance, we can express 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* using the following rule (also called a signature):

$$os(Ip, win2k) \vee os(Ip, winXP) \longleftarrow tcp(Ip, \_, \_, \_, yes, syn, 128).$$

Obviously, we are not limited to a single atom on the right hand side, and thus we can easily specify a signature using multiple packets (stimulus-response for instance). Moreover, since the set of packets seen so far corresponds to the facts of the logic program, the passive OSD module is naturally endowed with a memory. These two characteristics help to improve upon the current state-of-the-art passive tools.

Using answer set programming over more classical logic programming (e.g. Prolog) is advantageous: sound and complete semantics, expressivity (disjunction in the head and both weak and strong negation), non-monotonicity, ability to do planning, etc. Yet, a major concern when using such an expressive language could be the time complexity. However, with careful management of the facts[1] it is possible to avoid most of the combinatorial explosion associated with the answer set semantics.

2. Active Module

Even though the use of logic programming allows to enhance significantly the accuracy of passive OSD tools, there is still a fundamental limitation: passive means waiting for the facts to come in. An active module could be of great help in some situations. The idea here is to use the deductions made by the passive module (discard some OS) and then generate a sequence of tests to actively gather the missing information. To avoid the drawbacks of usual active OSD tools, planning is used to ensure that only relevant tests will be executed and to avoid, as much as possible, performing tests that generate abnormal traffic.

The initial state describes the set of current possible OS (some may have been discarded due to passive deductions). The actions are the available active tests. The action effects encode the way an active test will partition the set of all OS depending on the test result (e.g. *Windows* hosts react like this, *Linux* hosts reacts like that, etc.). The goal is a state where only one given OS remains possible[2]. The planning task is thus to find "the best" sequence of tests to go from the initial state to the goal state.

3. Results

Early experiments with a prototype for the passive module show extremely promising results. Even at an early stage, the prototype clearly outperforms state of the art passive OSD tools in terms of accuracy.

[1] We keep one set of facts for each host and we use a mechanism to eliminate facts while keeping the knowledge they convey.

[2] We are currently working on encoding other goals.