Programming Research Group

APPLYING ILP TO THE LEARNING OF INTRUSION STRATEGIES

Steve Moyle and John Heasman

PRG-RR-02-03

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Intrusion detection is the identification of potential breaches in computer security policy. The objective of an attacker is often to gain access to a system that they are not authorised to use. The attacker achieves this by exploiting a (known) software vulnerability by sending the system a particular input. Current intrusion detection systems examine input for syntatic signatures of known intrusions. This work demonstrates that logic programming is a suitable formalism for specifying the semantics of attacks. Logic programs can then be used as a means of detecting attacks in previously unseen inputs. Furthermore ILP can be used to induce detection clauses from examples of attacks. Experiments of learning ten different attack strategies to exploit one particular vulnerability demonstrate that accurate theories can be generated from very few attack examples.

1 Introduction

With the increasing connectivity of academic and commercial networks via the Internet, more computer systems have become exposed to attacks designed to breach their security. Here an attack is considered to be a malicious transaction between the attacker and a service that is running on the target computer system. At its lowest level a transaction can be considered as a sequence of bytes. For an attack[1] to succeed the service must be vulnerable (to the particular attack), and the transaction must successfully exploit the vulnerability.

The current main technique for detecting inappropriate transactions is that of scanning for known signatures. This is a purely *syntactic* measure which does not consider the potential *operational semantics* of the transaction (within the context of the service and computer system). A syntatic rule set consists of rules of the type: `IF [transaction] CONTAINS [signature] THEN ACTION`, where the action may be to alert the system administrator or log the transaction. Once a signature is in place, an operationally equivalent transaction (i.e. an attack) – but with different encoding – will not be detected by the syntatic rule set.

In this work a semantic description of a malicious transaction is developed. With such a description, it can be deduced that syntactically different transactions can have the same operational semantics. That is two attacks can appear significantly different, but have identical effects on the system. This work introduces a semantic detection rule set representing rules of the type: `IF [transaction's opratational semantics] MATCH THOSE OF ANY [malicious operational semantics] THEN ACTION`.

This work empirically demonstrates that it is possible to induce updates to the semantic detection rule set using Inductive Logic Programming, by utilizing examples of detected attacks as input examples. Furthermore, the number of examples required to induce accurate rules is reassuringly low, given their likely (low) frequency of being detected.

The remainder of the paper is organised as follows. Section 2 introduces the main intrusion detection paradigms and explains in some detail a particular type of attack

[1] The terms *attack* and *exploit* are used interchangeably in this paper.

which is common. Section 3 outlines a logic based encoding to represent the operational semantics of malicious transactions (as byte sequences), which become available for use as background knowledge in inducing updates to the detection rule set by ILP techniques. In section 4, an experiment to re-generate detection rule sets is described, and the results and conclusions are presented in sections 5 and 6.

2 Intrusion detection paradigms

An intrusion is defined as *"any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource"* [12]. The two most common Intrusion Detection models are **anomaly** and **misuse** detection. Intrusion detection (ID) models can be characterised by performance metrics such as the proportion of **false positive**[2] and **false negative**[3] detections that an implementation produces as well as its susceptibility to **subversion errors**[4]. Although different implementations of an ID model are likely to have varying false positive and false negative proportions, the range of values is largely fixed by the model itself [4].

2.1 The Misuse Model of Intrusion Detction Systems

In the misuse model the intrusion detection system (IDS) has some knowledge of suspicious behaviour and it looks for transactions that match this knowledge. The knowledge is typically represented as a set of signatures, where each signature encapsulates the salient features of a different attack or class of attack. The primary advantage of this model is that the proportion of false positive detections is low and this can be reduced by lengthening each signature (i.e. specifying each attack in more detail[5]). A drawback of this model is that the proportion of false negative detections can be high, depending on the limitations in the knowledge contained within the list of signatures. This model provides no mechanism for automatically updating its knowledge and therefore requires human intervention to keep the signature list up-to-date. For this reason subversion errors are minimal, but not eliminated completely. An attacker may know a certain action causes a false positive and may repeat this action multiple times. The administrator might be wrongly tempted to remove the signature that causes this false positive instead of specialising it further. If he does so, the attacker can then launch the exploit that the defunct signature was intended to detect, effectively generating a false negative.

It is also difficult to encode contextual and operational information in the misuse model. This leads to a type of false positive detection in the sense that an attack against a particular architecture or operating system is detected while the target system is hosted on neither, rendering the attack benign. Many misuse IDS do not provide suitable syntax for associating certain signatures with specific targets. Systems that implement the

[2] A false positive is defined as a *legitimate* transaction that is incorrectly flagged as an *intrusion*.

[3] A false negative is defined as an *illegitimate* transaction that is regarded as a *legal*.

[4] A subversion error is defined as a successful attempt by an attacker to manipulate the system such that the proportion of false negatives is increased.

[5] This is analogous to *specialising* a rule in ILP.

misuse model (e.g Symantec NetProwler, and Snort) typically only provide basic pattern matching features. This means that an attacker is also able to bypass an IDS by obfuscating a known attack slightly so that it no longer matches a signature. This highlights a major flaw with this model in that signatures are stored and matched against transaction in only a *syntatic* manner.

2.2 The Anomaly Model of Intrusion Detection Systems

The other common intrusion detection framework is **anomaly detection**. Here the intrusion detection system has knowledge of normal behaviour and looks for exceptional activity since it is assumed that *"exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations could be detected from abnormal patterns of system usage"* [8]. Normal behaviour is defined in terms of metrics recorded when the system is running routinely. If the IDS is concerned with specific user activity then the set of metrics may include the number of processes running and the number of files open. If the IDS is monitoring a system as a whole then the set of metrics may include CPU usage, the number of network connections open, and sequences of system calls. Systems implementing anomaly-based detection have used propositional and other machine learning techniques (e.g. neural networks [7]) to determine whether an intrusion is taking place. However their suitablity has tended to depend on the environment in which the system operates and the set of metrics that is used.

The main advantage of the anomaly-based model is the low proportion of false negative detections made. This low rate can be reduced further by increasing the sensitivity to changes in the metrics. Unlike the misuse model, the anomaly model can potentially identify previously unseen attacks. In addition, once the system has been trained, it requires less human intervention since it is able to automatically suggest updates to its knowledge. This however, means that subversion errors are a serious threat. It may even be possible for an attacker to weaken the system from the start by attempting intrusions whilst the system is learning normal behaviour. The principle disadvantage of this model is the high proportion of false positive detections, since an intrusion may always generate abnormal behaviour but the reverse does not hold.

Most commercial intrusion detection systems (e.g. Cisco NetRanger, Internet Security Systems RealSecure) implement the misuse model since it requires little customisation for the environment it operates in. It is also favoured because of its low proportion of false positive detections. As Frandsen notes in his guidance on security incident tracking [10], the usefulness of an intrusion detection system is severly reduced if it produces so much information that serious intrusions cannot be identified in reasonable time.

2.3 An example of an attack: the buffer overflow

The Program Analysis study [3] and the Research Into Secure Operating Sytems project [1] identify a class of vulnerability relating to incomplete input validation – the most common form of which is the **buffer overflow**. The circumstances in which a buffer overflow can occur are well understood and wholly avoidable if the software is developed

Arbitrary data	Sequence of idle instructions	Exploit code	Mutiple copies of return address

Figure 1: Components of a malicious transaction containing a buffer overflow attack. The input may contain some arbitrary data at the beginning which can simply be ignored. This is "junk" that is also subject to the null byte problem discussed in Section 3.1.3.

according to one of many secure programming checklists [11, ch. 23]. There are numerous publications [13, 6, 9, 26] that suggest modifications to development and runtime environments, designed to detect vulnerable code and to constrain buffer overflow attacks. However, few of these systems are in widespread use, often due to the high overheads associated with them [6] or other limitations that they place on normal system operation. Without runtime measures to protect against buffer overflow attacks, the need for intrusion detection is increased.

2.3.1 A Technical Description of a Buffer Overflow attack

The concepts behind exploiting a buffer overflow vulnerability are simple but the implementation details are subtle. They are discussed thoroughly from the perspective of the attacker in the seminal paper *"Smashing the Stack for Fun and Profit"* [2]. Sufficient background and implementation details are reproduced here so that the reader may understand how such attacks can be detected.

Buffer overflow attacks exploit a lack of bounds checking on a particular user-supplied field within a transaction. The attacker supplies an overly large input transaction to the service. In the course of processing the transaction, the service copies the field into a buffer of a fixed size. As a result, memory located immediately after the buffer is overwritten with the remainder of the attacker's input. The memory that is overwritten may have held other data structures, the contents of registers, or, assuming the system is executing a procedure, the address from which program execution should continue once the procedure completes. It is the latter that is of most interest to the attacker.

The function that copies data from one location to another without regard for the data size is referred to in the sequel as simply *the dangerous function*. Before the service executes the procedure containing the function call that leads to the overflow the *return address* is pushed onto the stack so that program execution can continue from the correct place once the procedure is complete. Register information is then pushed onto the stack so that the original state can be restored. Space for variables local to the procedure is then allocated.

The attacker aims to overwrite the return address thus controlling the flow of the program. If he is unable to do this, he may still be able to overwrite values in the register save area; this may or may not allow him to execute his exploit code, depending on later instructions contained in the program. A malicious transaction input of a buffer overflow attack consists of several components as shown in Figure 1.

The dangerous function call takes the attacker's input and overwrites, in this order, the intended buffer: 1) the space allocated for the procedure's local variables and the register save area, and 2) the return address. The purpose of the sequence of *idle* instructions at the beginning of the exploit code is discussed in Section 3.1.3. The attacker aims to set the return address to a location within the idle sequence so that his exploit code is executed correctly. He supplies multiple copies of his desired return address to compensate for the uncertainty of where the actual return address is located. Once the procedure has completed, values from the (overwritten) register save area are popped off the stack back into system registers and program execution continues from the new return address – now **executing the attacker's code.**

For the attacker to correctly build the malicious field within the transaction, he must know some basic properties of the target system, including the following:

- The *processor architecture* of the target host as machine code instructions have different values in different processors.
- The *operating system* the target is running as each operating system has a different kernel and different methods of changing between user and kernel mode.
- The *stack start address* for each process which is fixed for each operating system/kernel.
- The approximate *location of the stack pointer.* This is a guess at how many bytes have been pushed onto the stack prior to the attack. This information together with the stack start address allow the attacker to correctly calculate a return address that points back into the idle sequence at the beginning of the exploit code.

The work described in the sequel is limited to the discussion of buffer overflow exploits for the Intel x86 family of processors running the Linux operating system. The application of the detection techniques presented in this paper to other processors and operating systems is discussed in the conclusions (Section 6).

3 The operational semantics of a buffer overflow attack as a logic program

This section develops background knowledge containing information about buffer overflow attacks. For this purpose it is useful to formulate a general approach for representing attacks against any class of vulnerability in a semantic manner. Such a formulation is effectively a knowledge engineering task, further classiffed as **level one** of Michie's knowledge approach[6]. The representation process is outlined below.

1. Determine the domain of the class of vulnerability. That is, the set of attributes, relations and facts from which the attack can be described. This may include representations of network or system protocols as well as information about machine instructions.

[6]Michie's knowledge approach [15] concerns itself with problem solving via knowledge representation and application. **Level one** is *"the transfer (by [the] programmer) of algorithmic knowledge from [a] book and [the] programmer into [a] program."*

2. Determine any constraints that narrow the domain. These are often inherent to the particular kind of attack.
3. Determine the goal of the attacker and express this in an appropriate form (e.g. in English).
4. Use a top-down approach to repeatedly reduce the goal(s) into subgoals, until each subgoal can be represented in a suitable manner. For example as a sequence of instructions in a protocol or low-level language. The use of real world examples can be helpful in deciding how to subdivide goals.

3.1 An English representation of a buffer overflow attack

The buffer overflow attack can be represented according to the steps defined above.

3.1.1 Domain

The domain of the buffer overflow attacks is the set of machine-code instructions on the target architecture (Intel x86). Although the exploit code can feasibly contain any valid opcode (subject to the constraints discussed in Section 3.1.3), there is only a small subset of instructions that the attacker is likely to use. This subset is likely to include register manipulation instructions (e.g. to move a value from one register to another), stack operations (to push and pop values on and off the stack), and basic arithmetic operations (e.g. to increment a register). The domain also includes the set of commands on the target that the attacker may have to use to accomplish his goal, and operating system specifics such as how system calls are set up and executed.

3.1.2 The goal of the attacker

Tanenbaum [25] notes that one user identifcation, called the super-user (or root), *"has special power and may violate many of the protection rules [set by the operating system]...Many ordinary users devote considerable effort to trying to find flaws in the system that allow them to become super-user without the password"*. This is often achieved by exploiting a vulnerable service in order to execute code that gives access to a command interpreter, known in Unix-based operating systems as a *shell*, from which other commands can be issued. In the sequel the term *shell code* will be used interchangeably with *exploit code* to refer to the part of the exploit that contains the instructions that the attacker wishes to execute.

3.1.3 Constraints

The *null byte problem* [2] states that the exploit code will not contain any bytes whose value is zero. This is because the functions that allow buffer overflows to occur – typically string operations in the C programming language such as *strcpy* and *strcat* – treat a *zero byte* as a string terminator. If the exploit code contains a zero byte, any bytes after it will not be operated on. This constraint has other implications. The sequence of characters representing the shell to be executed (e.g. */bin/sh*) cannot be terminated which means the exploit code itself must itself generate and place a zero byte after it in memory so that it can be treated as a string.

Furthermore, the exploit code must contain the relevant instructions to execute the *exec* system call. This is because the goal of the attacker (to open a shell) is predetermined. In the Linux operating system, system calls are carried out by placing the call number in the register *%eax* (and appropriate parameters in subsequent registers *%ebx*, *%ecx* and so on) then triggering software interrupt 0x80 which causes the system to jump into kernel mode. The system call number of *exec* is standard across all Linux kernels.

The set of instructions in the exploit code should not contain any hard-coded addresses since this will greatly reduce the chances of the exploit succeeding. The only hard-coded address in the complete attack must be the return address. Since the address of the sequence of characters representing the shell is required, the exploit code must contain instructions to calculate and then indirectly load this into a register.

The previous constraints can be classified as hard constraints since they specify what must and what must not occur in the input. It is also possible to define soft constraints, that is, conditions that cannot be stated in a precise manner. An example of a soft constraint concerns the length of the exploit code. Typically there is a limited amount of memory address space for the exploit code to occupy before the return address is reached. If the exploit code is larger than the available space, the return address will be set to the value represented by the four bytes of instruction code that happen to have been copied there and the attack will fail. The vulnerable service will almost certainly crash but the attacker will not accomplish his goal. A length constraint therefore states that the size of the exploit code must be smaller than the amount of memory before the return address. Of course, the latter may only be estimated and so this constraint can be paraphrased as the attacker must write the exploit code to be as compact as possible.

A final soft constraint is that the exploit code is likely to be prefixed by a number of instructions that have no purpose except to *idle* the processor. Idle instructions are used to reduce the uncertainty of guessing the correct return address. If there are no idle instructions the return address must be set to the beginning of the exploit code which requires a perfect knowledge of the state of the process' stack. By including a sequence of idle instructions at the start of the exploit code, the return address need only be set to an address within this sequence which greatly improves the chances of an exploit succeeding. This sequence of idle instructions is often referred to as a *NOP sledge*. *NOP* is the opcode that represents a *no operation* instruction within the CPU.

The number of idle instructions that prefix the exploit code may vary depending on the attacker's knowledge of the layout of the stack. Early buffer overflow exploits only used the *NOP* opcode in creating a sequence of idle instructions. This, however, is easily detectable by syntactic pattern matching. Recent exploits [20] use instructions that are deemed to be idle within the context of the exploit. An example of an idle instruction therefore, might be some arithmetic operation on a register which is not referenced in the exploit code. The attacker may also decide to use multibyte instructions as idle instructions (i.e. the opcode and operand take up more than a single byte). This reduces the chances of the exploit working, since the return address must now contain the address of the first byte of a multi-byte instruction, but if the circumstances are such that the attack can be repeated multiple times then this method can be successful[7].

3.1.4 Summary of constraints

- There will be a sequence of idle instructions, due to the uncertainty (of guessing the return address) constraint.
- There will be code to indirectly load the address of the shell string into a register, as there can be no hard-coded address. This code will be near the start of the exploit code.
- There will be code to set up the parameters then jump into kernel mode due to the Linux method of executing a system call. This will make up the last few instructions in the shell-code.
- The code prior to the setting up of the system call will null-terminate the sequence of characters representing the shell. This involves sub-steps of generating a zero byte inside a register and moving the zero byte to the address of the end of the sequence of characters representing the shell.

3.1.5 Representing the subgoals

When expressing the subgoals of an attack it is useful to express instructions in assembly language, a low level symbolic language in which instructions correspond to sets of machine language instructions.

Some example shell code, taken from *"Smashing the Stack for Fun and Profit"* [2] is shown in Figure 2 together with a disassembly and commentary. This example satisfies each of the constraints above and has been used as the basis for many real world exploits. Such is its versatility that it can be used in almost any Intel x86 Linux buffer overflow vulnerability. This example was used as a basis for the buffer overflow representation.

3.2 A Logic programming representation of a Buffer Overflow attack

The English description of the buffer overflow class of vulnerability can be expressed in the Logic programming language, PROLOG[8] [14]. Some of the PROLOG predicates that constitute the background information known about buffer overflow attacks are shown in the appendix. The procedural meaning of the predicates is that the first argument is the sequence of bytes received, in the form of a list, and the second argument is the suffix of the first list of bytes, the prefix part having been *consumed* by the predicate. Any other arguments represent additional parameters (e.g. register numbers and offsets)

[7]Often local exploits can be repeated mutiple times whilst remote exploits cannot. If a remote exploit fails, the server daemon must be restarted by the system administrator before the exploit can be attempted again.

[8]For the syntax of PROLOG refer to Clocksin and Mellish [5].

```
  0xeb,0x1f                   jmp 0x1f              jump to call
  0x5e                        popl %esi             pop address of string into %esi
  0x89,0x76,0x08              movl %esi,0x8(%esi)   place address of string
  0x31,0xc0                   xorl %eax,%eax        generate null long in %eax
  0x88,0x46,0x07              movb %eax,0x7(%esi)   terminate string
  0x89,0x46,0x0c              movl %eax,0xc(%esi)   place null long
  0xb0,0x0b                   movb $0xb,%al         set system call number
  0x89,0xf3                   movl %esi,%ebx        move address into %ebx
  0x8d,0x4e,0x08              leal 0x8(%esi),%ecx   load address of address
  0x8d,0x56,0x0c              leal 0xc(%esi),%edx   load address of null long
  0xcd,0x80                   int $0x80             jump to kernel mode
* 0x31,0xdb                   xorl %ebx,%ebx        generate null long in %ebx
* 0x89,0xd8                   movl %ebx,%eax        move null long into %eax
* 0x40                        inc %eax              increment %eax
* 0xcd,0x80                   int $0x80             jump to kernel mode
  0xe8,0xdc,0xff,0xff,0xff    call -0x24            call pop instruction
  /bin/sh                     .string "/bin/sh"     shell string
```

Figure 2: Common Intel x86 Linux machine code with disassembly to open a shell. Instructions marked with an asterisk are not strictly required. However, they allow the shell-code to terminate cleanly via the `exit` system call if something fails or once the attacker closes the shell. If they are omitted, the offsets in the `jmp` and `call` opcodes must be altered appropriately.

that the predicate requires to consume the input. Note that some of the predicates in the background knowledge have equivalent rules in a syntatic rule set of a typical IDS[9].

```
% rule1 - Checks for whether list of bytes contains command
% rule1(+[Byte_sequence]).
rule1(Bytes):- contains_command(Bytes).

% rule2 - Checks for idle sequence of length 128
% rule2(+[Byte_sequence]).
rule2(Bytes):- idle_sequence(Bytes,128).
```

The `rule1/1` predicate is equivalent to a set of rules, each pattern matching a different command. The `rule2/1` predicate however, cannot be represented succinctly in a syntatic rule set. It detects sequences of idle instructions, where an idle instruction is defined by the predicate `idle/3`. The syntatic rule set may have a signature for detecting a sequence of NOP opcodes as shown in Figure 3 (a) but if the NOP opcode is interspersed with FWAIT[10] instructions (Figure 3(b)) then the signature will fail, while the `rule2/1` predicate however, will succeed. The set of predicates allow a more thorough representation of the situations than may be found in syntatic rule set. However,

[9]This section refers only to misuse model IDS.

[10]This opcode causes the processor to check for and handle pending floating-point exceptions before proceeding. In the context of a buffer overflow attack, this operation can be regarded as an idle instruction.

0x8B	0x0C	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x10	0x0A

(a) NOP sequence detected by a typical syntatic rule set.

0x8B	0x0C	0x90	0x9B	0x90	0x9B	0x90	0x9B	0x90	0x9B	0x10	0x0A

(b) NOP & FWAIT sequence detected by semantic rule set. A typical syntatic rule set will not detect the sequence and will generate a false negative.

Figure 3: Alternative *NOP Sledges*: a) utilising only a sequence of the `NOP` instruction; and b) using a sequence combining both `NOP` and `FWAIT` instructions.

used disjointly, the semantic rule set is likely to produce a false positive detection rate comparable to that of a symantic rule set. This rate can be lowered by combining facts using conjunctions of predicates, as in `rule3/1`.

```
% rule3 - Checks for a sequence of at least 128 idles and
% checks whether list of bytes  contains command
% rule3(+[Byte_sequence]).
rule3(Bytes):- idle_sequence(Bytes,128), contains_command(Bytes).
```

Contextual information can easily be encoded in Prolog. The example predicate below encapsulates basic information about the operating system that a specific host is running, the processor type and any services that are operational (a more complete representation may include the name, version and patch level of each service).

```
% host_info/4 - Contains useful information about a specific host
% host_info(+Name, +Processor, +Operating system, +[Services]).
host_info(workstation1, intel_x86, linux, [ftpd, sshd]).
host_info(server1, dec_alpha, windows, [ftpd, httpd]).
host_info(workstation2, sparc, solaris, [sshd]).
```

The `host_info/4` predicate can be used in conjunction with the other predicates in the background knowledge to provide context-specific intrusion detection semantic rules.

```
% rule4 - Check target is Intel and check for sequence of 128 idles
% rule4(+Host,+[List_of_bytes]).
rule4(Host, Byte) :- host_info(Host,intel_x86,_,_) ,
                     idle_sequence(Byte, 128).
```

A semantic intrusion detection rule set created from propositional predicates, as introduced above, could typically lead to performance comparable to a syntatic rule set (in terms of rate of false positive detections), although the false negative detection rate is likely to be lower. The context dependent nature of the semantics of a buffer overflow attack motivate the use of first order logic, which reduces the rate of false positive detection. This leads to a predicate as shown below (which covers semantic variations in the exploit code taken from *"Smashing the Stack for Fun and Profit"* [2]).

```
rule5(A,I) :- consume_idles(A,B),
              consume_load(B,C,Addr,Len),
              consume_place_addr(C,D,Addr,Len),
              consume_null_long(D,E,Null,Addr),
              consume_null_operation(E,F,Len,Null,Addr),
              consume_set_sys_call(F,G,Null,0x0b,Addr),
              consume_params(G,H,Addr,Null,Len),
              consume_system_call(H,I).
```

The disadvantage of using the above predicate in a semantic rule set is that attacks that have different *structural* properties are not detected. Structural variations on the attack strategy above are given in Figure 1. A simple solution is to add more rules that cover variations in the exploit code structure but this requires substantial human effort. If the expert is faced with new real world examples he believes can be expressed using the background information that is already encoded, he might ask *"is there a way of automating this process?"*. This is the motivation for the following section, which demonstrates that inducing updates to the semantic rule set is possible.

4 An experiment to induce semantic protection rules

This section describes controlled experiments to (re-)learn semantic intrusion detection rules using the ILP system ALEPH [22]. The motivation for the experiment can be informally stated as testing the hypothesis that *ILP techniques can recover reasonably accurate rules describing attack strategies.*

The basic method was to use correct and complete PROLOG programs to randomly generate sequences of bytes that successfully exploit a buffer overflow[11]. Such byte sequences are considered as *positive examples*, E^+, of attacks. The byte sequence generating predicates were then removed from the PROLOG progam leaving a common background program to be used as background knowledge, B_{IDS}, to ALEPH. This background program was then used, along with postive examples of attack byte sequences, to induce rules similar to the byte sequence generating predicates. The induced rules could then, in theory, be used in the semantic rule set introduced in Section 3.2.

Ten variations in buffer overflow attack strategy were investigated. Three of these are shown in Table 1 together with an informal verbal description of each, the target PROLOG rule and an example byte sequence.

The controlled experiments were carried out as follows:

1. Select a stategy, S_i, for exploiting a buffer overflow.

[11] All such generated sequences, when supplied as part of a transaction with a vulnerable service, enabled the attacker to open a shell.

Table 1: Three variations of buffer overflow attack strategies. For each variation a description of the strategy is provided, along with a PROLOG clause to detect an instance of the strategy, and one example instance.

Strategy Number	Strategy Description, a PROLOG clause to detect it, and an Example

1	This strategy starts with idle instructions, performs a jmp-call-pop subsequence to get the address of the string then puts the string address in the required place. It then generates a null long which it uses to terminate the string and complete the array. The system call number is then set up (parameter 1), then the remaining parameters are set up before the system call is executed.
	
	`strategy1(A,I) :- consume_idles(A,B),`
	`consume_load(B,C,Addr,Len),`
	`consume_place_addr(C,D,Addr,Len),`
	`consume_null_long(D,E,Null,Addr),`
	`consume_null_operation(E,F,Len,Null,Addr),`
	`consume_set_sys_call(F,G,Null,0x0b,Addr),`
	`consume_params(G,H,Addr,Null,Len),`
	`consume_system_call(H,I).`
	
	`strategy1([153,248,252,245,155,155,252,252,235,29,95,153,137,127,`
	`10,144,102,49,192,152,155,136,71,9,137,71,14,176,11,137,251,141,`
	`79,10,141,87,14,205,128,232,222,255,255,255,/,b,i,n,/,b,a,s,h],[]).`

2	This strategy is a variation on strategy 1. Instead of putting the address of the string in the required place as soon as the address is obtained, the null long is generated and used. Parameters are then set up and the system call executed as in strategy 1.
	
	`strategy2(A,I) :- consume_idles(A,B),`
	`consume_load(B,C,Addr,Len),`
	`consume_null_long(C,D,Null,Addr),`
	`consume_null_operation(D,E,Len,Null,Addr),`
	`consume_place_addr(E,F,Addr,Len),`
	`consume_set_sys_call(F,G,Null,0x0b,Addr),`
	`consume_params(G,H,Addr,Null,Len),`
	`consume_system_call(H,I).`
	
	`strategy2([249,155,252,144,144,249,155,235,22,91,49,192,136,67,7,`
	`137,67,12,137,91,8,176,11,141,75,8,141,83,12,205,128,232,229,255,`
	`255,255,/,b,i,n,/,s,h],[]).`

3	This strategy is different from strategies 1 & 2 in that the first operation after the idles is to set up the null long. Next it generates the string address via jmp-call-pop (this time the jmp-call-pop operation cannot simply choose any register in which to load the string address). The rest of the strategy is the same as in strategy 1.
	
	`strategy3(A,I) :- consume_idles(A,B),`
	`consume_null_long(B,C,Null,Addr),`
	`consume_load(C,D,Addr,Len),`
	`consume_place_addr(D,E,Addr,Len),`
	`consume_null_operation(E,F,Len,Null,Addr),`
	`consume_set_sys_call(F,G,Null,0x0b,Addr),`
	`consume_params(G,H,Addr,Null,Len),`
	`consume_system_call(H,I).`
	
	`strategy3([248,248,155,245,155,186,255,255,255,255,66,235,24,95,137,`
	`127,9,136,87,8,137,87,13,49,192,4,11,137,251,141,79,9,141,87,13,205,`
	`128,232,227,255,255,255,/,b,i,n,/,a,s,h],[]).`

This strategy is encoded as a PROLOG clause.

2. Generate Examples.

   Given the particular strategy and the common background knowledge B_{IDS} (encoded as a PROLOG program), examples of particular malicious transactions were randomly generated (deductively) to produce a set of examples $E^+_{S_i} = \{e^+_{(1,i)} \dots e^+_{(m,i)}\}$ (as lists of bytes).

   Each of these postive examples was tested on a vulnerable service[12] to confirm that they indeed were successful attacks.

   A population of *random* examples $E^?$ was generated by randomly permuting true positive examples.

3. Produce a learning curve for recovering rules for S'_i.

   Induce semantic intrusion rules from a varying number of randomly selected positive examples[13] The rules were produced using `ALEPH` in positive-only mode [17].

   The accuracy of the rules were assessed with respect to the remaining positive examples.

4.1 Results

The learning curves for the recovery of each of the ten intrusion strategies are presented in Figure 4. Here, accuracy is measured as the proportion of test set examples which are detected by the recovered rule. The proportion of the test set examples which are *not* detected by the recovered rule is the intrusion detection *false negative* rate.

It can be seen from Figure 4 that, on average, for the attack strategies studied, relatively few examples of attacks are required to recover rules that detect a high proportion of attacks. In fact, for all of the ten studied attack strategies, only ten positive attack examples were required to recover a rules that were 100% accurate on the test set. This finding is discussed in the following section.

5 Discussion

The graphs in Figure 4 demonstrate that it is possible to induce a semantic detection rules for buffer overflows attacks that are 100% accurate. Furthermore, these results can be obtained using, on average, a low number of examples for the induction. The need for few examples can be attributed to the relatively low size of the hypothesis space, which was a result of the high input and output connectedness of the background predicates. The estimated size of the hypothesis space for the recovery of each attack strategy is presented in Table 2 and is comparable to the classic ILP testbed problem, the *Trains* [24]. Srinivasan provides estimates of the hypothesis space for the trains problem and other common ILP data sets in [21].

[12] A dummy application created by the authors for test purposes.

[13] The random example pool remained fixed across all runs.

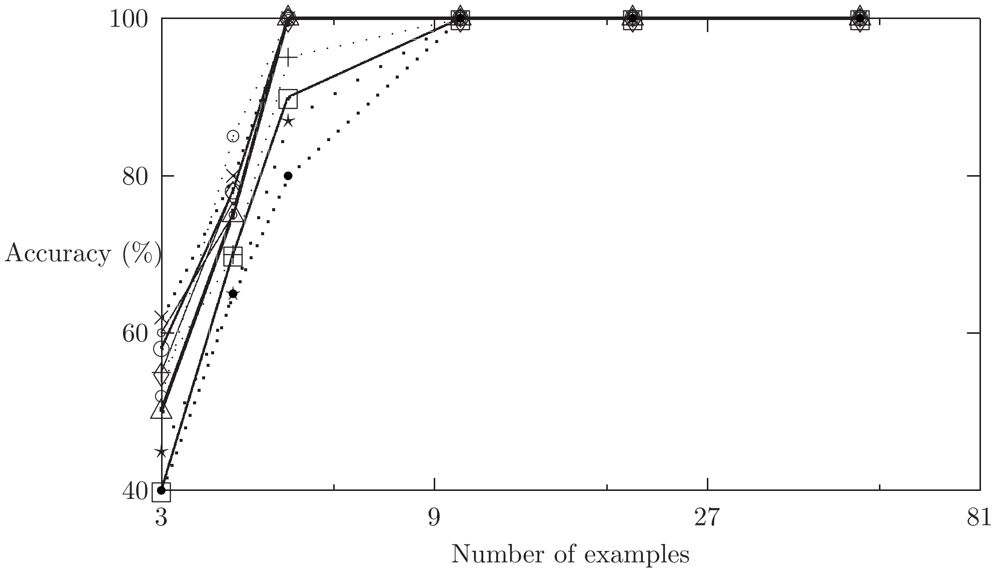

Figure 4: Learning curves for the recovery of ten different intrusion strategies from examples. Each point on the graph represents the average accuracy of 10 runs.

Table 2: Table of the estimated hypothesis space for the ILP recovery of each variation of attack strategy (three of which are presented in Table 1). Estimates of the hypothesis space were produced by `ALEPH`.

Sequence number	Estimated Size of the Hypothesis space
1	850
2	660
3	630
4	910
5	740
6	810
7	980
8	830
9	790
10	770

Examples of the clauses recovered for each of the experiments can be found in Appendix [SAM / JH].

This result demonstrates that ILP was well suited to the intrusion problem studied. It may well be suited to real world intrusion detection problems, provided the background theory is sufficient. These experiments indicate that the background information was highly accurate and relevant. The importance of relevancy has been studied by Srinivasan and King [23]. This application domain is very "crisp" and well defined. In a particular context, a byte op-code *always* performs a particular function. For example when the byte `0x9B` is executed on an Intel x86 processor it *always* triggers the `FWAIT` functions to be performed. Such well defined, and cleanly specifed domain is well suited to being represented in a logic program and susceptable to ILP. Contrast this with a biological domain where the activity of a particular system varies continously with respect to many inputs – for example the concentration of certain chemicals.

6 Conclusions

The work described has shown that it is possible to encode the operational semantics of attack transactions in a logical framework, which can then be used as background knowledge to an ILP system. Sophisticated attack strategies can then be represented in such a framework.

In studying the particular application of a buffer overflow attack, it has been shown that ILP techniques can be used to learn rules that detect – with low false negative rates – attack strategies from relative few examples of attacks. Furthermore, only positive examples of attacks are necessary for the induction of attack strategy rules.

6.1 Future work

There are many areas for futher research including broadening the scope of the existing work, and applications of the techniques to other domains. Some of these are suggested below.

Further experiments could be performed to focus on reducing the accuracy and completeness of the background theory. One might claim this would be a more representative model of the real world, however it is worth remembering that buffer overflow attacks have been studied in great detail and that manual analysis of shell-code is certainly possible, albeit time-consuming (this is in contrast to other applications of ILP where the relevancy of some background information can only be assumed). The experiments described in this paper were carried out free from misclassification errrors. Further experiments may investigate the effect of *noise* on the learning process since it is feasible in a real application of this system that an expert might misclassify some examples. If misclassification errors have a severe effect on the learning process, they might be eliminated by stipulating that each example is tested in some form of sandbox[14] before classification.

[14] A sandbox is a protected, limited environment where unforeseen application behaviour does not risk damage to the rest of the system.

Future work may consider controlled experiments on different buffer overflow attack strategies. A progression of this work would be to focus on encoding sufficient background knowledge to learn attack strategies for buffer overflows on other operating systems (e.g. versions of BSD, Solaris, Microsoft Windows) and other processor architectures (e.g. Sparc, Dec Alpha). The background knowledge could also be extended to correctly detect shell-code that contains more complex features such as socket operations to enable communication with the shell to occur over the Internet, and also self-decrypting code[15].

Future work may also consider other classes of vulnerability. The encoding of suitable background knowledge for many attacks against higher level protocols such as FTP or HTTP, or (database) languages such as SQL is likely to require less effort since these attacks are platform independent (and therefore only require a common set of background predicates).

Another area of interest is the application of the semantic detection rule set generated by ILP to a real world environment. This would enable comparison with other intrusion detection systems. The reliability of the semantic system, its ability to handle high volumes of traffic, and its computational requirements are likely to be largely dependent on the underlying PROLOG system.

An approach similar to the one developed in this work might be applicable to biological domains – particularly that of functional genomics. There has been substantial development of technology to generate gene sequence data for many organisms. From the nucleotide sequences of the genes it is possible to deduce the amino acid sequences of the proteins which they encode. In [18] it is stated that

> A significant challenge in the analysis and interpretation of genetic sequence data is therefore the accurate recognition of patterns within the data that are diagnostic for known structural or functional features within the protein.

They go on to describe efforts to use ILP techniques to induce grammatical representations to tackle the challenge. The nucleodtide sequences in genes (and their counterpart amino acids in the proteins) can be seen to be analogous to the byte codes of transactions in this work. However, the biological domain is less crisp than that of micro-processor architectures. For instance the contextual information in a biological system will likely include concentrations of chemical components, rather than simply the length of bytes.

One logic programming issue to investigate might be the possibility to recode the background knowledge of the IDS predicates in a more compact way so that statements similar to the following can be expressed: *all strategies that include a null byte production sequence in a contigous block*. This is a form of higher order logic programming and is might be achieved using similar techniuqes to those in [19].

Finally, it is worth noting that although this work is preliminary, the results are encouraging, with many research avenues to explore.

[15] This is an advanced method of IDS evasion. The attacker places a decoder at the start of the exploit code and subsequent instructions – that may be detectable by a syntatic rule set – are supplied in an encrypted form then decoded in realtime.

Acknowledgements

This worked benefited from the generous assistance of Ashwin Srinivasan in the use of his `ALEPH` system. Steve Moyle was supported in part by the EU project Sol-EU-Net, IST-11495.

References

[1] R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, D. Webb *Security Analysis and Enhancements of Computer Operating Systems*. NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, 1976.

[2] "Aleph One". *Smashing The Stack For Fun And Profit*. Phrack 49, 1996.

[3] R. Bisbey II, D. Hollingworth. *Protection Analysis Project Final Report*. ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, 1978.

[4] G. Bruneau *The History and Evolution of Intrusion Detection*. The SANS Institute, `http://rr.sans.org/intrusion/evolution.php`, 2001

[5] W. F. Clocksin, C. S. Mellish. *Programming in Prolog*. 3rd ed. Springer-Verlag, Berlin, 1987.

[6] C. Cowan, C. Pu et al. *StackGuard: Automatic Adaptive Deception and Prevention of Buffer-Overflow Attacks*. Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 1998.

[7] H. Debar, M. Becker, D. Siboni. *A neural network component for an intrusion detection system*. In Proceedings of the IEEE Symposium on Research in Security and Privacy, pages 1-11, May 1992.

[8] D. Denning. *An intrusion-detection model*. IEEE Transactions on Software Engineering, 13(2):222-232, 1987.

[9] H. Etoh, K. Yoda. *GCC extension for protecting applications from stack-smashing attacks*. IBM Research Division, Tokyo Research Laboratory, `http://www.trl.ibm.com/projects/security/ssp/`, 2000

[10] M. Frandsen. *Incident Tracking*. The SANS Institute, `http://rr.sans.org/intrusion`, 2000.

[11] S. Garfinkel, G. Spafford. *Practical UNIX and Internet Security*. Sebastopol, O'Reilly and Associates, 1996.

[12] R. Heady, G. Luger, A. Maccabe, M. Servilla. *The architecture of a Network Level Intrusion Detection System*. Technical Report CS90-20, Department of Computer Science, University of New Mexico, 1990.

[13] D. Larochelle, D. Evans. *Statically Detecting Likely Buffer Overflow Vulnerabilities* . Department of Computer Science, University of Virginia, 2001.

[14] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1993.

[15] D. Michie. *On machine Intelligence*. Ellis Horwood, Chichester, 1986.

[16] S. Muggleton. *Inverse Entailment and Progol*. New Generation Computing, 13(3 and 4):245–286, 1995.

[17] S. Muggleton. *Learning from positive data.* In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, p. 225-244, 1996.

[18] S.H. Muggleton, C.H. Bryant, A. Srinivasan, A. Whittaker, S. Topp, and C. Rawlings. *Are grammatical representations useful for learning from biological sequence data? – a case study.* Computational Biology, 8(5):493-522, 2001.

[19] S. Muggleton and C. D.Page. *Beyond first-order learning: inductive learning with higher-order logic.* Technical Report PRG-TR-13-94, Oxford University Computing Laboratory, University of Oxford, 1994.

[20] F. J. Serna. *Polymorphic Shellcodes vs. Application IDSs.* NGSEC Research Team, `http://www.ngsec.com/docs/polymorphic_shellcodes_vs_app_IDSs.PDF`, 2002.

[21] A. Srinivasan. *A study of two probabilistic methods for searching large spaces with ILP.* Technical Report PRG-TR-16-00, Oxford University Computing Laboratory, University of Oxford, 2000.

[22] A. Srinivasan. *The Aleph Manual.* `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`, 2001.

[23] A. Srinivasan, R. D. King. *An empirical study of the use of relevance information in Inductive Logic Programming.* Technical Report PRG-RR-01-19, Oxford University Computing Laboratory, University of Oxford, 2001.

[24] R. E. Stepp, R. S. Michalski. *Conceptual Clustering: Inventing Goal-Oriented Classifications of Sturctured Objects.* In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors,*Machine Learning: An Artificial Intellingence Approach*, p. 738. Morgan Kaufmann, 1986.

[25] A. Tanenbaum *Operating Systems Design and Implementation.* Prentice-Hall International, Englewood Cliffs, 1987.

[26] D. Wagner, J. Foster, E. Brewer, A. Aiken. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.* Proceedings of the 2000 Network and Distributed Systems Security Conference, pages 3-17, University of California, Berkeley, 2000.

Appendix

This appendix presents sample PROLOG code that provides the basis for the background knowledge for the intrusion detection system, B_{IDS}.

```
% Notes:-
% Any assembler mnemonics are given in AT&T syntax.
% A long is 32 bits, a word is 16 bits, a byte is 8 bits.

% Auxiliary Predicates

% decode/2 - Decodes a register byte, checks src and dest regs are the same
% decode(+Byte,-Register).
decode(B,R1) :- nonvar(B),
        R1 is (B /\ 2'00111000) >> 3,
        R2 is B /\ 2'00000111,
        R1 =:= R2,!.
```

```
% Instruction Predicates

% idle/3 - Predicate representing an idle instructions
% idle(+Hex_opcode,+Assembler_mnemonic,+ID).
idle(0x90,'nop',0).
idle(0xfc,'cld',1).
idle(0xf9,'stc',2).
idle(0xf5,'cmc',3).
idle(0xf8,'clc',4).
idle(0x99,'cltd',5).
idle(0x9b,'fwait',6).

% command/2 - Predicate representing a command
% command(+Command_name,+ID).
command(['/','b','i','n','/','s','h'],0).
command(['/','b','i','n','/','b','a','s','h'],1).
command(['/','b','i','n','/','c','s','h'],2).
command(['/','b','i','n','/','t','c','s','h'],3).
command(['/','b','i','n','/','a','s','h'],4).
command(['/','b','i','n','/','b','s','h'],5).

% Predicates to consume input

% consume_system_call/2 - Consume a system call.
% consume_system_call(+[List_of_bytes],-Remainder).
consume_system_call([B1,B2|Rest],Rest) :- system_call(B1,B2,_).

% consume_null_long/4 - Consume a null long generator
% consume_null_long(+[List_of_bytes],-[Remainder],-Null_register,+Addr_register).
consume_null_long([B1,B2|Rest],Rest,Null,Addr) :- null_long(B1,_,_),
                    decode(B2,Null),
                    lessthan(Null,4),
                    not(equals(Addr,Null)).

% Intrusion Detection Predicates

% idle_sequence/2 - Find the maximum number of
% consecutive idles in the list of bytes
% idle_sequence(+[List_of_bytes],-Max_idles).
idle_sequence(Bytes,MaxSequence) :- sequence(Bytes,MaxSequence,0,0).

sequence([],Max,_,Max).
sequence([Byte|Rest],Final,Current,Max) :- idle(Byte,_,_),
                   plus(1,Current,NewCurrent),
                   greater(NewCurrent,Max,NewMax),
                   sequence(Rest,Final,NewCurrent,NewMax).
sequence([Byte|Rest],Final,Current,Max) :- not(idle(Byte,_,_)),
                   sequence(Rest,Final,0,Max).
```

```
% contains_command/1 - Is true if the list of bytes contains a command
% contains_command(+[List_of_bytes]).
contains_command(Bytes) :- command(Command,_),
                           concat(_,B2,Bytes),
                           concat(Command,_,B2).
```