Masquerade Detection via Customized Grammars

Mario Latendresse
Volt Services/Northrop Grumman
FNMOC U.S. Navy
latendre@iro.umontreal.ca

February 8, 2005

Abstract

We show that masquerade detection, based on sequences of commands executed by the users, can be effectively and efficiently done by the construction of a customized grammar representing the normal behavior of a user. More specifically, we use the Sequitur algorithm to generate a context-free grammar which efficiently extracts repetitive sequences of commands executed by one user – which is mainly used to generate a profile of the user. This technique identifies also the common scripts implicitly or explicitly shared between users – a useful set of data for reducing false positives. During the detection phase, a block of commands is classified as either normal or a masquerade based on its decomposition in substrings using the grammar of the alleged user. Based on experimental results using the Schonlau datasets, this approach shows a good detection rates across all false positive rates – they are the highest among all published results known to the author.

1 Introduction

Masquerade detection is probably the last protection against such malicious activity as stealing a password. Anomaly detection, based on the user's behavior, is one of the primary approach to uncover a masquerader. It can be done using data from various sources, ranging from sequences of *commands* (a.k.a *programs*) executed by the user to sequences of system calls generated from the user's activities. In this study, we use sequences of programs executed by the user in a Unix environment. These programs are either explicitly called by the user or implicitly called via other programs (e.g. scripts). Our experimental results are based on the Schonlau datasets [6] which, as we will see in Section 3, have both classes of programs.

In masquerade detection, the normal behavior of a user should be represented by a *user profile*. It is typically built during the *training* phase, done offline – a *training* dataset, free of masquerade attacks, should be available to do it. The *masquerade detection phase*, where attempts are made to classify the behavior of the alleged user, is done online and once the training is completed. We can partition the *user profiles* in two classes: *local profiles* where the normal behavior of a user is solely based on the user's data; and *global profiles* where the normal behavior of a user is also based on additional data – typically from other users. The local profiles are usually simpler to implement than the global ones. On the other hand, the local profiles may have less capability at masquerade detection. In our work we use global profiles.

We can further partition the classes of masquerade detection approaches in two subclasses: approaches that either update or do not update, during the masquerade detection phase, the user profile. This update could be partial, for example by being only local: only the behavior of the user has any impact on its profile. In our work we use partial updating of the global profiles. This simplifies the implementation and deployment of our approach.

In this work, we demonstrate that the Schonlau datasets have many repetitive sequences of commands among users and in each training dataset. We believe that this is typical of Unix systems where common scripts are shared among the users. For each user training data, we use a linear time algorithm, called Sequitur, to extract the *structure* of these repetitive sequences in the form of a context-free grammar. We also compute local and global statistics for these sequences. From the grammars, we also extract the repetitive sequences having a minimum frequency and length. These sequences are considered to be scripts that are shared among users – we call them *global scripts*.

Section 3 motivates our approach by an analysis of the Schonlau datasets. Section 4 presents the main technique used by our approach and its experimental results are in Section 5. Section 6 presents some inferior variations of the main method. The analyzes of some incorrect classifications are done in Section 7. In Section 8 we discuss the computational cost of our approach. We summarize other published methods in Section 9. To make our paper self contained, we review the Sequitur algorithm in the next section.

2 The Sequitur Algorithm

The Sequitur algorithm was created by Nevill-Manning and Witten [4] to extract hierarchical structures from strings. It constructs a context-free grammar based on one string: the language of that grammar contains only that string. The construction of the grammar is efficient as it can be done in linear time on the length of the string. We will briefly describe this algorithm and state one important property relevant for our detection algorithm.

2.1 A review of the Sequitur algorithm

Recall that a context-free grammar is a quadruple (S, N, Σ, P) where Σ is the set of terminals, N the set of nonterminals (N and Σ do not intersect), S the start symbol ($S \notin N \cup \Sigma$), and P the set of production rules of the form $n_k \rightarrow x_1 x_2 \dots x_n$ where $x_i \in N \cup \Sigma$, $n_k \in N \cup \{S\}$. The nonterminal n_k (or S) is the left-hand side (lhs) of the production rule and $x_1 x_2 \dots x_n$ is its right-hand side (rhs). We will call the production rule with lhs S, the *main production*; all other productions are *auxiliary productions*. Notice that in this study, the Unix commands form the set Σ.

Let $C = (c_i)$ be the string of elements $c_i \in \Sigma$ from which a Sequitur grammar will be created. The grammar is initialized with the main production $S \rightarrow c_1 c_2$, where c_1 and c_2 are, in that order, the first two elements (e.g. commands) of C; they are removed from C. In general, Sequitur proceeds sequentially on C by adding to the end of the rhs of the main production the next command of C not yet added. New productions will be created and deleted by maintaining the following two constraints on the current grammar.

Unique Digram No digram, i.e. pair of adjacent terminals or nonterminals, occurs more than once (without overlap) across all rhss of the grammar.

Useful Production Any nonterminal occurs more than once across all the rhss of the grammar.

The constraint *Unique Digram* has a tendency to create new production rules whereas the constraint *Useful Production* removes some. In most cases, a repeated digram occurs when adding an element of C to the end of the rhs of the main production. A new production rule $n_k \rightarrow x_1 x_2$ is created if a digram $x_1 x_2$, where $x_i \in \Sigma \cup N$, repeats in the rhss of the grammar and the digram is not the rhs of any existing production. The lhs n_k replaces the repeated digram. If the digram already exists as the rhs of a production, the lhs of that production simply replaces the repeated digram. A production with lhs n_k is removed if n_k does not occur more than once in all rhss of the grammar; if it occurs once, the rhs of that production replaces n_k – in other words, n_k is *inlined*. This is another case where a repeated digram can be created.

Figure 1 presents two examples of grammars generated by the Sequitur algorithm. Lower case letters are terminals and upper case letters are nonterminals – i.e. we do not use Unix commands in these examples. There are no relations between the nonterminals of G_1 and G_2. Terminals are added to the

Generation of Grammar G_1 from input string `dadabfbfeaeabgbg`	**Generation of Grammar G_2 from input string** `bcabcaca`
S → `dada`	S → `bcabc`
S → AA A → `da`	S → A`a`A A → `bc`
S → AA`bfbf`	S → A`a`A`a`
S → AABB B → `bf`	S → BB B → A`a`
S → AABB`eaea`	B → `bca` (A inlined)
S → AABBCC C → `ea`	S → BB`ca`
S → AABBCC`bgbg`	S → BBC B → `b`C C → `ca`
S → AABBCCDD D → `bg`	

Final grammar G_1	**Final grammar G_2**
S → AABBCCDD A → `da` B → `bf` C → `ea` D → `bg`	S → BBC B → `b`C C → `ca` (deleted: A → `bc`)

Figure 1: **Two examples of the Sequitur algorithm applied to the strings `dadabfbfeaeabgbg` (left) and `bcabcaca` (right).**

main production (i.e. S → ...) until a repeated digram occurs. We step through every time a digram is replaced by a nonterminal (i.e. when a digram repeats) or a production rule is inlined/deleted. For example, for G_1, when the digram `da` occurs twice in the main production, the new production A → `da` is created. For G_2, when the rule B → A`a` is created, the rule A → `bc` becomes useless – therefore it is deleted and inlined in B → A`a`. As a matter of fact, for grammar G_1, only the constraint *Unique Digram* had to be enforced, but both constraints had to be enforced for G_2.

2.2 Relevant properties

The following proposition should now be obvious:

Proposition 1 (Repetition) *The expansion of any auxiliary production rule, from the generated* Sequitur *grammar of string C, is a substring that occurs more than once in C.*

Notice that since the grammar generates exactly the string C, the expansion of the main production cannot repeat in C. In other words, the last proposition does not apply to the main production – this is the main reason to treat it differently than the auxiliary production rules.

This simple proposition is the basic element of our approach: the grammar can be used to represent some repeated sequences of the input data C – the training data in the context of masquerade detection. Indeed, not all repeated sequences are extracted from C. That is, the converse of this last proposition is not true: There are repeated non-overlapping substrings of C that may not be the expansion of any production of the Sequitur grammar. This is obvious once we consider that any proper substring of the expansion of an auxiliary production repeats in C, yet it is not the expansion of that production. It is not even the case that a repeated substrings in C will necessarily be the *substring* of the expansion of an auxiliary production. For instance, for G_1 in Figure 1, the substring `ab` repeats in the input string, yet it is not the substring of the expansion of any auxiliary production. Despite this fact, a large number of repeated sequences are substrings of the expansions of auxiliary production rules.

The Sequitur algorithm not only generates a grammar that mostly represents the repetitive sequences, it does so recursively. That is, repetitive sequences that occur inside or across longer ones have their own production rules. For example, this is apparent in grammar G_2 of Figure 1 where the digram `ca` is repeated across two productions, the main one and in production B. This sort of repetitive structures does occur in the context of executed commands since scripts may be embedded inside other scripts.

3 Motivation of Our Approach

Schonlau *et al.* [6] have made available some datasets for the study of masquerade detection algorithms. They are available at `www.schonlau.net`.

These datasets are based on the commands executed by 70 users of a multi-user Unix systems. The `acct` auditing facility was used to collect the commands. Actually, `acct` records the *programs* executed and not the commands directly typed by the users – more on this below – but to remain consistent with the documentation of the Schonlau datasets, we still use the term *commands* to refer to the executed programs. Among the 70 users, 20 were used as potential masqueraders and 50 as potential victims of masquerades. The data from the 20 masqueraders are not explicitly available. For each of the 50 users, 5000 commands can be assumed to be from the legitimate user. They are used as *training data*. For each user, 10000 more commands are provided, divided in 100 blocks of 100 commands: each block either comes from the legitimate user or from one of the 20 masqueraders – this is the *testing data*. This is done with a known uniform random distribution, but we should not use that knowledge during training or detection of masquerades. Among the 50 users, 29 have at least one masquerade block.

There are many long common substrings (i.e. sequences of commands), among users, in the training sets as well as in the testing sets. In all likelihood, many were generated by executing scripts – i.e. commands that usually execute several programs without the user intervention. In fact, the technique used to collect the sequences of commands (i.e. the `acct` auditing facility) does record the programs executed – not the commands typed directly by the users.

For example, user 16 has a sequence of 35 commands – see Figure 2 – which occurs more than 20 times in its training data. Such a sequence of commands can hardly be taken as directly typed by the user, but is more likely emitted by a script.

In general, the Schonlau training datasets contain hundreds of long sequences (i.e. more than 10 commands) repeated more than ten times. The generations of the 50 Sequitur grammars, presented in the next section, clearly demonstrate the existence of these sequences. We believe that this is not a peculiarity of the Schonlau datasets but rather an aspect of the way programs are composed on Unix systems.

```
getpgrp LOCK true ls sed FIFO cat date generic
generic date generic gethost download tcpostio
tcpostio tcpostio tcpostio cat generic ls generic
date generic rm ls sed FIFO rm UNLOCK rmdir
generic tcppost sh LOCK
```

Figure 2: **A sequence of 35 commands occurring 20 times in the training data of user 16.**

In summary, the large number of repetitive sequences indicates an important aspect:

> Many repetitive sequences of commands are probably *not* directly typed by the users but produced by scripts which are explicitly called by the users. We conclude that the profile of a user should be based on those repetitive sequences.

The main problem is to discover those repetitive sequences. This motivates our approach presented in the next section.

4 Our Approach

In this section we present the techniques used in our approach to represent the normal behavior of a user – i.e. its profile – and detect masqueraders.

4.1 Constructing the user profile

For each user, a Sequitur grammar is generated based on the uncontaminated sequence of commands C (e.g. the sequence of 5000 commands for the Schonlau datasets). As it was shown in Section 2, the production rules represent repetitive sequences of commands. For each production rule, we compute the total frequency of its expansion in C. This can be efficiently done since the frequency of each lhs (non-terminal) is maintained during the generation of the

Production Rules	Frequencies	
	User	**Others**
A → B C	4	0
B → `cat mail csh`	42	231
C → D E	12	0
D → F `java`	22	0
E → `csh make`	14	0
F → G `java`	33	0
G → `java_wr` H `base` I I `egrep`	42	0
H → J `dirname`	45	1545
I → `egrep egrep`	84	1126
J → `expr expr`	50	1762

Table 1: **Excerpt of production rules for the grammar of user 1.**

grammar[1]: The total frequency is computed recursively by taking into account the productions where the lhs occurs.

For each production, besides the total frequency, we compute the frequency of its expansion across all other user training data – this is the *across frequency*.

We also compute the global set of scripts used by all users. It is the expansion of all production rules that occur at least five times among all users. This is used by our detection algorithm to reduce the negative impact of unseen commands that, we believe, are actually part of an unseen script (see the next section for its usage).

The production rules themselves are not very important, it is rather their expansion, and their associated frequencies, that are used during the detection of masquerades. For example, it would be acceptable, and more efficient, for our detection algorithm to represent the set of expansions in a trie; although the Sequitur algorithm is an efficient means to discover some repetitive sequences. We did not implement the trie mechanism since we are not emphasizing the efficiency of the implementation.

[1]The frequency of a lhs is maintained to apply the *Useful Production* constraint – if the frequency of the lhs falls below two, the production must be inlined and removed from the grammar (see Section 2).

Average number of rules	260.9
Average length of the expansions	11.4
Average frequency of the expansions	15.7
Maximum frequency over the 50 users	1664
Maximum length over all expansions	486

Table 2: **Statistics for the 50 Sequitur grammars.**

Table 1 presents an excerpt of the Sequitur grammar of user 1. The entire grammar is much larger and cannot be easily presented. For each production, two frequencies are displayed: the frequency of the expansion of that production in the training data for user 1, and its frequency in the training data for the 49 other users. For example, the expansion of J (i.e. `expr expr`) occurs 50 times in the training data of user 1, and 1762 times in the training data of all other users. Table 2 presents some statistical facts for the 50 grammars constructed by the Sequitur algorithm based on the 50 users training data.

Another part of the training phase is the determination of a constant for the evaluation function of a block during the detection phase. This part is described in Sub-section 4.3.

4.2 Detection of Masquerades

The Schonlau datasets have, for each of the 50 users, a sequence of 10000 commands which might be contaminated in block of 100 commands by some other users. Therefore, in the following explanation the detection algorithm is described on a block of commands.

Let G be the grammar of the alleged user for the block to be classified. The classification of a block is based on its evaluation and a global threshold. If the value, obtained from the evaluation, is larger or equal to the threshold, the block is considered normal; otherwise it is considered a masquerade. The evaluation of a block is done by sequentially breaking it into substrings which are expansions of some production rules of G. In general, during the overall evaluation of a block, we have a set of segments of the block not yet matched with any production rule

of G. An expansion of a production rule of G which is a substring of a segment is a candidate to break that segment. We use the following evaluation function e, on productions p, for which their expansions are substrings of at least one of the current segments of the block.

$$e(p) = l_p \frac{f_p}{f_p + \frac{F_p}{k}} \tag{1}$$

where l_p is the length of the expansion of production p, f_p the frequency of the expansion of the production, F_p its across frequency, and k a constant. The next subsection motivates the form of that equation and describes our technique to determine a good value for k – a search that is done offline during training.

The production p_0 that gives the largest value is removed from the segment: this either eliminates completely the segment, generates two other segments, or only one.

The previous process is repeated on all current segments of the block until no more segments contain a substring which is the expansion of some production rule of G. Let F be the set of productions found by that process, then $\sum_{p \in F} e(p)$ is the base value of the block.

The remaining segments may contain previously unseen commands from G. If a segment contains a global script as a substring, the unseen commands of that global script are counted as one unseen command. That is, a value of one is subtracted from the base value for each global script found, and their unseen commands are not considered individually.

For the remaining unseen commands, their frequency, with a maximum of 4, is subtracted from the base value of the block. Based on experimental results, it does not change substantially the evaluation if the frequencies are not taken into account, that is, if a value of -1 is given to each unseen commands.

Notice that the value of $e(p)$, according to Equation 1, cannot have a value larger than l_p – e.g. for a block of 100 commands, its value cannot exceed 100.

4.3 Determining a value for k

In Equation 1, the value k serves as an averaging factor for the across frequency F_p. In fact, if we were assuming $k = 49$, the expression $\frac{F_p}{k}$ would be the average frequency of the expansion of production p among the 49 other users. Actually, the main intention of that expression is to compare the frequency f_p to the across frequency F_p taking into account the number of other users. But it is not clear that the value 49 is the right one – its value should be determined during the training phase for all users.

Essentially, the technique we have used to determine k is the following – it was done for each integer value from $k = 1$ to $k = 49$, picking the best result. For each user, ten blocks of 100 commands are randomly selected from each other users training data. In the case of the Schonlau datasets, 490 blocks are selected for each user. The evaluation of each block is done according to the method of the last section. The lowest total, across all users, is considered the best. For the Schonlau datasets the best value for k is 7. In the section on variations of our method (see Section 6), we also show the detection results – a ROC curve – when using the extreme value 49. The results are in agreement with this procedure: the overall performance of the detection is better with $k = 7$ than with $k = 49$; although for very low false positive rates, the value $k = 49$ is better.

4.4 Updating the user profile

During the detection phase, the profile of the user is modified if the block is classified as normal. The Sequitur algorithm is applied – using the new normal block as input – to modify the user grammar. This would usually extend the grammar by adding new production rules. The frequencies of the production rules are modified accordingly, but the across frequencies are not modified; and the global scripts set is not extended. In other words, only the local aspect of the profiles of the users are maintained, not their global aspect; this greatly simplifies the application of our approach in a distributed environment.

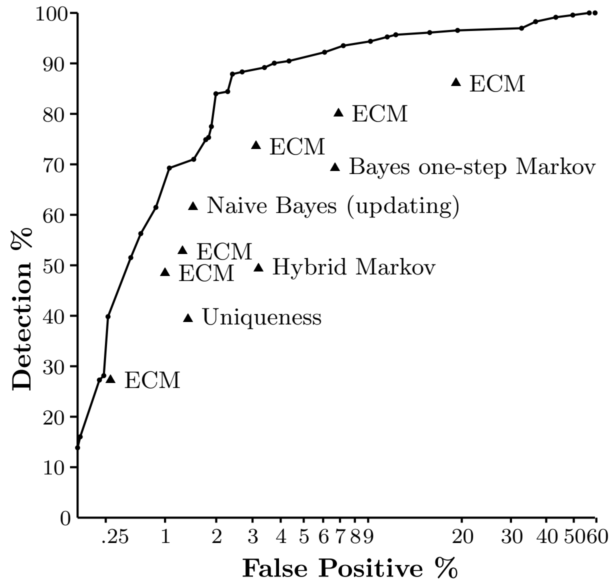

Figure 3: **ROC curve for our main method, for $k = 7$. The x-axis is logarithmic. Also included are some best-outcome results (triangles) of other good performing methods.**

5 Experimental Results

Figure 3 presents the main results of our approach using a Receiver Operating Characteristic curve (ROC curve). This shows the relation of the false positive rates versus the detection rates of masquerades. We have also included some best-outcome results for some other good performing methods (these results were taken from [5, 6]). Notice that the x-axis is logarithmic since we prefer to have a more precise view of the detection rates for false positive rates below 10%.

The ECM method of Oka *et al.* gives some of the best results previously published. Our approach detects even more masquerades at all false positive rates. To our knowledge, no published results based on the Schonlau datasets are better at any false positive rate.

6 Variations of our Method

In this section we present some further experimental investigations done on our main method. Three variations were tried: 1) with value $k = 49$; 2) no global scripts; and 3) only the frequencies of the commands are used, not the sequences. The last case also covers another variation to our main method, namely, to evaluate positively the already *seen* commands that are left out after decomposing a block during detection. Case 3 will show that this would diminish the detection rate.

6.1 With $k = 49$

This is a very simple variation to show that the technique used to determine k is successful on the Schonlau datasets. Figure 4 presents the ROC curves for both $k = 7$ and $k = 49$. We can see that for $k = 7$ the detection rates are higher for most of the false positive rates; although it is better for $k = 49$ when the false positive rates is below 0.3%. Still, the case $k = 49$ is a viable alternative superior to all other published methods.

6.2 No Global Scripts

This is a simple variation of our main method: no global scripts are used when evaluating unseen commands. The resulting ROC curve, compared to our main method with $k = 7$, is presented in Figure 5. The general tendency is an increase in false positives for the same rate of detection. There is clearly a decline of the detection rates around the 1% false positive rate compared to the main method *with* global scripts.

6.3 Command frequencies only

Our method is based on repetitive sequences of commands. This sub-section looks into a simpler version based on the frequencies of the commands for the user and across all users without taking into account their ordering. We apply a similar evaluation as function e (see Eq. 1). Namely, for each command c of a block we use the following equation where f_c is the

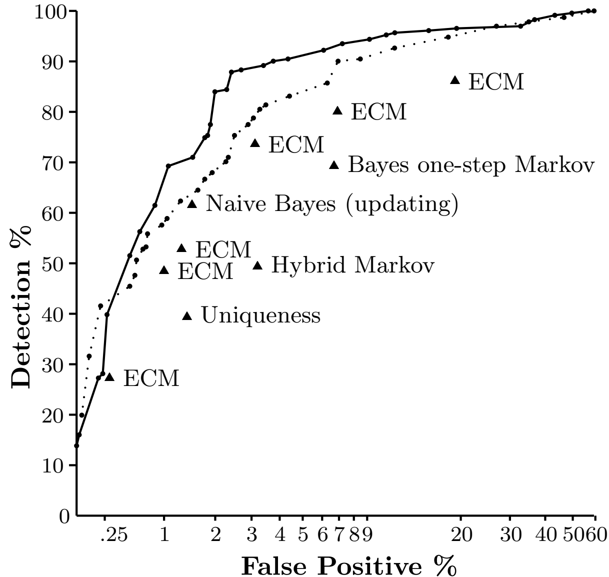

Figure 4: **ROC curve for our method for $k = 49$ (dotted line) compared to the determined $k = 7$ (solid line) .**

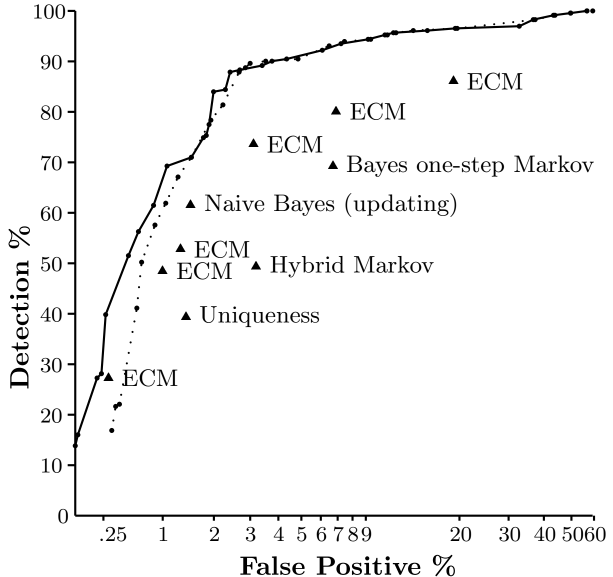

Figure 5: **ROC curves for our method without using the global scripts (dotted line) compared to the original main method (solid line).**

frequency of the command c in the training data of the user, F_c the across frequency of the command c among all other 49 users and k is a constant.

$$v(c) = \frac{f_c}{f_c + \frac{F_c}{k}} \quad (2)$$

We sum over all commands of the testing block resulting in one value. The frequencies, with a maximum of four, of the unseen commands in a block are negatively added to this value. As the previous method, one global threshold value is used to classify a testing block. Updating of the frequencies of the user, not the global ones, is also applied using that threshold.

Figure 6 presents the results for $k = 7$ by varying the threshold value from -4 to 70. ECM is better for at least one false positive rate and Naive Bayes is slightly better. This is also clearly inferior to the main method presented in the previous section. This shows that the ordering of the commands is important.

7 Failure Analyzes

In this section we analyze some of the erroneous classifications done by our approach – the main method with $k = 7$. We believe this shows the limit of our method but also of the difficulty of improving any method on the Schonlau datasets.

First, as a general view, Figure 7 presents histograms of false positives, false negatives and detected masquerades for different thresholds. These histograms give a general idea of the dispersion of false positives and negatives across users. It also gives a quick view of the users that appear problematic.

For false positives, at threshold 12, user 20 has a very large number of them compared to the other users. There is a total of 72 false positives, and 30 are generated by that user. If user 20 were taken out of the statistics at that threshold, the false positive rate would fall to 0.88% with the same detection rate.

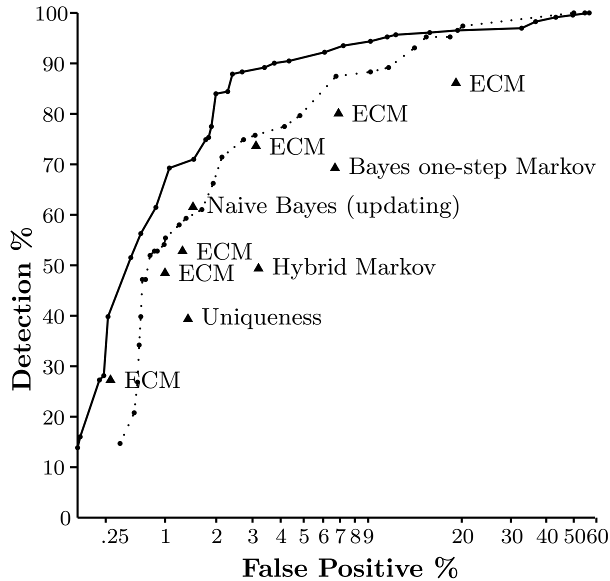

Figure 6: **ROC curves using the command frequencies with $k = 7$ (dotted line) compared to the original main method with $k = 7$ (solid line).**

7.1 False negatives

At threshold 23, user 12 has six false negatives – the largest number for that threshold. Its testing block 69 has the decomposition presented in Table 3; it is valued at 61.78[2]. It is the first false negative for user 12 with that threshold. More precisely, for thresholds 20 to 31 it has the value 61.68. The value may differ with other thresholds since the grammar is updated according to that threshold. Its value ranged from 55, with thresholds of 50 to 85, to 66.63, with thresholds of -2 to 19. Essentially, this block, as six others, evaluates to a high score across all thresholds despite being a masquerade. How can a masquerade gives such a high evaluation?

One substring of length 38 has a value of 33.25. By itself, this substring alone is enough to make it a false negative. The main observation: It occurs 3 times in the training data and 3 times for all other

[2]Recall that the maximum value of a decomposition is 100.

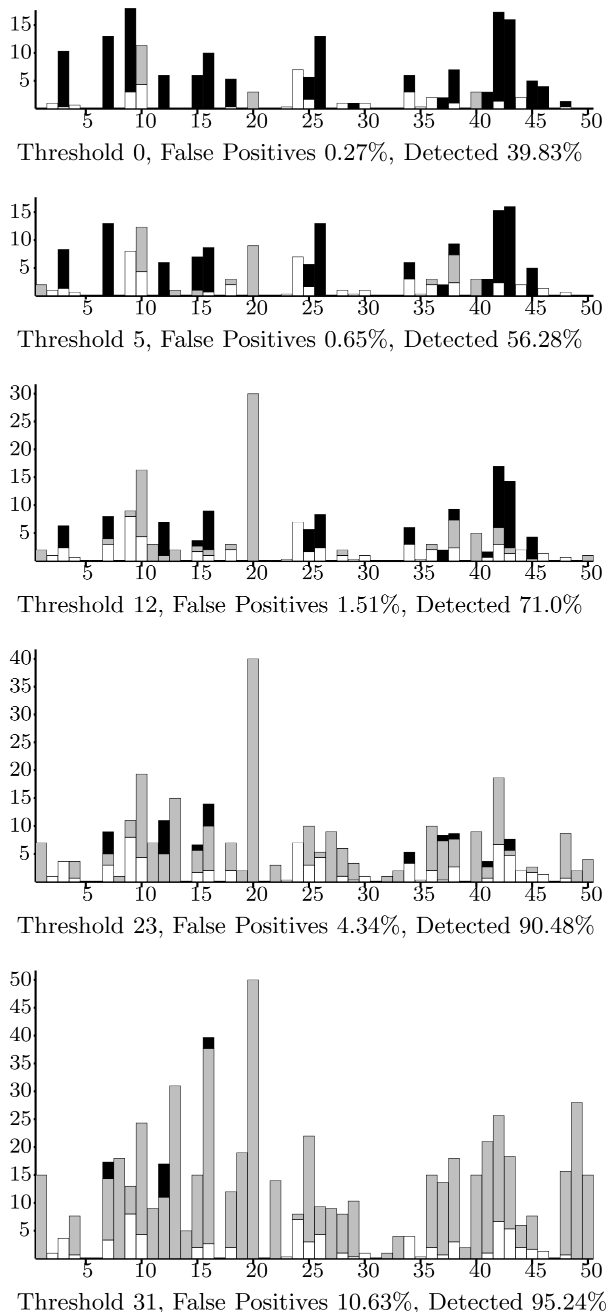

Figure 7: **Combined histograms of false positives (gray), false negatives (black) and detected masquerades (white). The x-axis represents users; the y-axis the number of blocks.**

Production Rules	$e(p)$	l_p	f_p	F_p
A → B B	33.25	38	3	3
H → I J	11.32	14	20	33
$X_1 \rightarrow X_2\ X_3\ K$	8.7	23	2	23
C → D E generic	3.61	8	10	85
X_5 → L M	3.36	6	2	11
F → G find	0.68	3	37	877
X_6 → ls generic	0.68	2	48	460
Skipped substrings: (cat generic) (cat generic ls)				

Table 3: **The decomposition of testing block 69 of user 12, a masquerade, that evaluates to 61.78. The X_i nonterminals were generated during the updating of the grammar.**

users. The evaluation function could be blamed: it offers no difference between a substring that occurs often or not for low frequencies across all users. Yet, this block, as six others, really appears as coming from the legitimate user.

7.2 False positives

The 46th testing block of user 20 is not a masquerade, although it is evaluated at -2.21. It is a false positive. Table 4 presents the decomposition of that block. Only three substrings of length 2 were found in the grammar. The rest of the block, which mainly contains the substring '`configure configure configure`', was skipped since no production expansions were substrings of it. Although, the command `configure` was just seen in the previous block. In order to give a higher value to this block, the individual commands should be taken into account. But as it was shown in the section on variations of our main method – for command frequencies only – this would have an overall adversed effect.

Table 5 presents the decomposition of the testing block 6 for user 49, a false positive. It has value 27.04 – not an extreme case as the previous block. As it can be seen from the values F_p, the reason for the low score is that the substrings of block 6 are common among other users. It is difficult to apply any *global* approach to avoid such a false positive.

Production Rules	$e(p)$	l_p	f_p	F_p
A → configur sed	1.59	2	2	25
A → configur sed	1.59	2	2	25
A → configur sed	1.59	2	2	25
Unseen commands: config.g(3), tr(18).				

Table 4: **Decomposition of testing block 46 of user 20. It is not a masquerade although its value is very low at -2.21. The block contains the substring `configur configur configur` numerous times. The command `configur` is first seen in the previous block which has a high evaluation of 80.8.**

Production Rules	$e(p)$	l_p	f_p	F_p
A → B C D E	14.77	38	2	22
$X_1 \rightarrow X_2$ F	6.08	10	10	45
G → H I	3.53	21	16	553
J → grep echo	1.94	2	5	1
K → L gethost	1.27	4	39	584
M → xwsh sh	0.48	2	5	111
Unseen commands: drag2(3)				

Table 5: **Decomposition of testing block 6 of user 49; its value is** 27.04**; it is not a masquerade.**

8 Computational Cost

The efficiency, or computational cost, of an anomaly detection algorithm is an important aspect. If it is very costly, it may become useless. Two phases should be considered for the efficiency of our method: the training phase and the detection (classification) phase.

For conducting our experiments, the implementations of the `Sequitur` and detection algorithms were done using the Scheme language. We compiled the code using the Bigloo 2.6c compiler on a Red Hat 9 system. All times reported are for a 2.26GHz, 1GB, Intel Pentium 4 computer.

The generation of the 50 grammars, for user 1 to 50, took 38.3 seconds: An arithmetic average of 765 milliseconds per user. This includes the time to read

the 5000 commands from a file. Some grammars took longer or shorter to generate. For example, grammar 30 took only 90 milliseconds to generate. This is due to the low number of generated production rules – only 42 compared to the average of 260. The average performance could easily be improved as there was no effort to implement an efficient Sequitur algorithm.

The classification of a block has two parts: its evaluation and the updating of the grammar. Over the 50 users and their testing blocks, that is 5000 blocks, the average time to evaluate and update for one block was 127 milliseconds. For user 30, the average was 40 milliseconds. Without updating, the average time to classify a block, over 5000 blocks, was 55 milliseconds.

9 Related Work

In comparing experimental results between methods, we believe it is important to take into account a major aspect: does the method use local or global profiles to represent the normal behavior of the users. A global representation is more complex to implement than a local one. Our method is global while some others reported in this section are local; although the updating of the profiles for our method is local.

Schonlau *et al.* [6] have reported the results of six methods: Hybrid Multi-step Markov, Bayes 1-step Markov, Compression, Uniqueness, Sequence-match, and IPAM. The experimental results are inferior to ECM for all false positive rates. For example, none of these methods, for the updating case, have a detection rate superior to 40% for a false positive rate of 1%. Our experimental results are superior to all of these.

Wang and Stolfo work [7] has the advantage of using a local representation for the normal behavior of a user. It is therefore not a surprise that we obtain better experimental results. Moreover, the main objective of that work was to demonstrate that a one-class training was as good as a two-class training approach.

Ju and Vardi [2] masquerade detection algorithm is based on rare sequences of commands. There is an instance of the algorithm for each length (e.g. 3, 5). They call their approach Local Command Pattern (LCP). We believe that the weakness of LCP is the lack of variable length sequences as used in our approach. They do not present any particular algorithm to discover the sequences as they are extracting all sequences of a fixed length. One of the best results is a false positive rate of 1.11% with a detection rate of 59.74% (for sequences of length 5).

Maxion and Townsend [3] have used the Naive Bayes approach on the Schonlau datasets. Such an approach is similar to the command frequencies method presented in Subsection 6.3 as it does not take into account the order of the commands but only the probabilities of occurrence of the commands. In general, a Naive Bayes classifier has to classify sets of observations among a set of categories. In this case, for each user u, there are two categories: user-u, or not-user-u. The probability of category user-u given the command c, is denoted $p(u|c)$. It can be evaluated using Bayes rule, namely $p(u|c) = \frac{p(c|u)p(u)}{p(c)}$. The probability of user u emitting command c, that is $p(c|u)$, was evaluated using $\frac{f_{c,u}+\alpha}{5000+\alpha A}$ where $f_{c,u}$ is the frequency of commands c for user u in its training data, A is the number of distinct commands in the training data, and α a small constant (e.g. 0.01). The category not-user-u can similarly be evaluated. The two probabilities are compared to classify a block of commands as either a masquerade or coming from user u. This approach is global as it refers not only to the user command frequencies but also to the frequencies for all other users. It also uses updating during detection. Their experimental results with the Schonlau datasets are good. For example, they report as one of their best results a false positive rate of 1.3% with a detection rate of 61.5%. Our method has a lower false positive rate, namely 0.85% (for $k = 7$), at such a detection rate. From their ROC curve published in [3], we conclude that our method has a higher detection rate for all false positive rates.

Coull *et al.* [1] use techniques from bioinformatics to detect masqueraders. Although the approach is innovative and the representation of the user behavior is local, the experimental results are not convincing. For example, they consider that one of their best results is a detection rate of 75.8% with a false positive rate of 7.7%; at such a detection rate, our method

has a much lower false positive rate, namely 1.8% (for $k = 7$).

Oka *et al.* [5] designed a method called Eigen Co-occurrence Matrix (ECM) for anomaly detection. It is based on Eigen face recognition techniques. The method is global but the experimental tests were done using only local profiles. The computational cost appears high but this is probably due to their implementation technique. The results obtained are the bests published for local profiles. We have extensively compared the results of this method with ours in the ROC curves of Sections 5 and 6 – our results are even better at all false positive rates. We believe this is mainly due to the global representation of our approach.

10 Conclusion

Our masquerade detection method based on repetitive sequences of commands was shown to be effective on the Schonlau datasets. As far as we know, the experimental results reported in this paper are superior to all published results based on the Schonlau datasets. More precisely – for all false positive rates – the detection rate is higher than all published methods, known to the author, for that datasets.

Our approach is quite efficient by using the Sequitur algorithm which is linear on the length of the training data. This could be completed with a more efficient data structure to store the discovered repetitive sequences.

Our method has the advantage of full control over the false positive rates. A unique global threshold can be varied to increase or decrease it – even below 1%.

We also believe our method naturally fits its environment. For instance, the global scripts correspond to a clear identifiable operational reality of the computing environment. If some of them were known, our algorithm could easily be improved by relying less on our heuristic to guess them.

11 Acknowledgments

Ms. Mizuki Oka kindly provided the data for the ROC curve of the ECM method.

References

[1] S. Coull, J. Branch, B. Szymanski, and E. Breimer. Intrusion detection: A bioinformatics approach. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 24. IEEE Computer Society, 2003.

[2] W. H. Ju and Y. Vardi. Profiling UNIX users and processes based on rarity of occurrence statistics with applications to computer intrusion detection. Technical Report ALR-2001-002, Avaya Labs Research, March 2001.

[3] R. Maxion and T. Townsend. Masquerade detection using truncated command lines. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-02), Washington, D.C.*, pages 219–228. IEEE Computer Society Press, June 2002.

[4] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[5] M. Oka, Y. Oyama, H. Abe, and K. Kato. Anomaly detection using layered networks based on eigen co-occurrence matrix. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection, (RAID), Sophia Antipolis, France, LNCS 3224*, pages 223–237. Springer, September 2004.

[6] M. Schonlau, W. DuMouchel, W. Ju, A. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. *Statistical Science*, 16(1):1–17, 2001.

[7] K. Wang and S. J. Stolfo. One-class training for masquerade detection. In *3rd IEEE Workshop on Data Mining for Computer Security (DMSEC 03)*, Nov. 2003.