Finding the Vocabulary of Program Behavior Data for Anomaly Detection

C. C. Michael *
Cigital Labs

Abstract

Application-based anomaly detectors construct a baseline model of normal application behavior, and deviations from that behavior are interpreted as signs of a possible intrusion. But current anomaly detectors monitor application behavior at a high level of detail, and many irrelevant variations in that behavior can cause false alarms. This paper discusses the preprocessing of audit data ultimately used by application-based anomaly detection systems. The goal is to create a more abstract picture of program behavior, filtering out many irrelevant details. Our specific approach automatically identifies repeating sub-sequences of behavior events and sequences of events that always occur together. Although this preprocessing technique can be used with a wide variety of program-based anomaly detectors, we present empirical results showing how it improves the performance of the well-known stide anomaly detection system.

1 Introduction

Host-based intrusion detection systems analyze data captured on a single information system. They might monitor users or software processes, for example. The data captured can be user data, such as keystrokes, login/logout times, operational profiles, or programs run during a session. Alternatively, the data can be program behavior data such as system calls or internal program states of monitored programs.

There is no "grand truth" of what constitutes and anomaly. Just as two humans may disagree about what constitutes an anomaly, so two anomaly detectors may disagree without either being technically right or wrong. Each anomaly detection system has a model of normal behavior that it uses as a baseline, and this model necessarily only captures some aspects of the state of the world. It is deviations from *this baseline* that are flagged as anomalies and potential intrusions. Therefore, we can hope to reduce false alarm rates by finding better models of normal behavior — ones whose definition of an anomaly corresponds to malicious behavior more often.

Thus, we would like to filter out those aspects of normal behavior that are irrelevant to the question of whether or not an intrusion is taking place. We would like to describe normal behavior at a more abstract level where benign variations are less likely to be seen. This paper describes one way of doing this.

We describe ways to automatically identify certain idioms of normal program behavior — idioms that can vary slightly without there being malicious intent. Our goal is to find a *vocabulary* for program behavior that captures the essence of that behavior without reflecting irrelevant details.

Normally, program behavior is captured in an *audit trail* that can be used by an intrusion detector to spot anomalies. Our approach is to preprocess the audit trail, simplifying it with the help of a vocabulary that has been deduced from training data. Thus, the audit trail that the anomaly detector eventually sees is more abstract, contains fewer unnecessary details, and is thus less likely to reflect benign variations in program behavior that might trigger false alarms.

2 Background

Some of the earliest work in intrusion detection was performed by [3] in the early 1980s. This paper defines an intrusion as any unauthorized attempt to access, manipulate, modify, or destroy information, or to render a system unreliable or unusable. Intrusion detectors are meant to warn of attempted intrusions.

Intrusion detection techniques are generally classified into two categories: anomaly detection and misuse detection. Misuse detection systems try to identify behavior pat-

*This work was sponsored under DARPA contract N66001-00-C-8056

terns characteristic of intrusions, but this can be difficult if an attack does not follow one of the patterns already known beforehand to characterize an attack On the other hand, anomaly detectors try to characterize the *normal* behavior of a system, so that any deviation from that behavior can be labeled as a possible intrusion. The work discussed in this paper deals with anomaly detection.

Anomaly detection assumes that misuses or intrusions are correlated to abnormal behavior exhibited by either a user or the system. Anomaly detection approaches must first determine the normal behavior of the object being monitored, then use deviations from this baseline to detect possible intrusions. The initial impetus for anomaly detection was suggested by [3], where it was noted that intruders could be detected by observing departures from patterns of use established for individual users. Anomaly detection approaches have been implemented in expert systems that use rules for normal behavior to identify possible intrusions [13], systems that establish statistical models for user or program profiles [5, 7, 15–18, 22], and systems that use machine learning to construct models of user or program behavior [4, 6, 8, 12].

Many intrusion detection techniques, like that of [7], characterize *application* behavior in terms of information system *audit data*. When an executing application requests an operating-system service (such as accessing files, allocating memory, and so on), this fact is recorded in an audit data stream that can be examined by an intrusion detector. First, the audit data is reduced to a series of symbols, with each symbol representing a different type of request such as `write`, `malloc`, `exit`, and so on. (The audit stream may also include information such as parameters, and return values, but that information is discarded.) The now classic approach of [7] works by moving a sliding window of length n across the symbolic audit data stream (for some n), creating a series of n-grams, each containing a series of n consecutive operating system requests. These n-grams are characterized as being either normal or abnormal, depending on whether they were seen previously in training data taken from non-intrusive program executions.

3 Vocabulary extraction for intrusion detection.

The goal of vocabulary extraction is to find a vocabulary that can be used to describe audit streams more abstractly. The idea is to start with an audit trace that consists of a series of symbols, and convert it to a sequence of *meta-symbols*, where each meta-symbol represents one or more consecutive symbols from the original trace. For example, the original audit stream

```
abcabcabcddeabcabcdf
```

might be converted to the more abstract execution trace

```
ABAdf,
```

where `A` represents one or more consecutive occurrences of the string `abc` and `B` represents the string `dde`. In this example, the `d` and `f` at the end of the trace are not represented by any vocabulary symbol, so they are copied verbatim.

Instead of using meta-symbols, it is also possible to replace vocabulary terms with simpler versions of themselves (this will be explained in more detail below). The advantage of this approach is that it simplifies the audit data without *adding* complexity by increasing the the number of audit symbols that the detector has to deal with.

Finally, the vocabulary extraction system described in this paper can be used as an intrusion detector in itself. In that capacity, it double-checks the results of other intrusion detectors and is often able to intercept false alarms.

The rest of this paper is organized as follows: in Section 4 we begin by discussing the implementation of a vocabulary extractor. Next, we discuss various ways of applying vocabulary extraction (Section 5), and then we present an evaluation of the system in Section 6.

4 Implementation of a Vocabulary Extraction System

The description of our approach will be simplified by some extra terminology. A *tandem repeat* is a string w that can be written as the concatenation of some other string v with itself, e.g., $w = vv$. A *tandem array* is a series of repeated substrings; a string is a tandem repeat if it can be written as some substring v concatenated with itself n times for some $n \geq 2$; e.g.,

$$w = v^n; \qquad n \geq 2. \tag{1}$$

The *tandem repeat types* of set of strings S are the strings that appear in tandem arrays in S. That is, the tandem repeat types are the set

$$\{v : (\exists s \in S, w \in s : w = v^n, n \geq 2)\}.$$

A *primitive tandem repeat type* is a tandem repeat type that does not contain any other tandem arrays.

Finally, a string w of symbols is *atomic* with respect to a set of exemplar strings S if and only if, for all strings $s \in S$ and all prefixes u such that $uv = w$, all occurrences of u in s are followed by v. Informally, if w is atomic then we never see any prefix of w without seeing all of w (at least as far as the exemplar strings are concerned).

Our goal is to find the atomic substrings and tandem repeat types in a set of strings S. To accomplish this, the execution traces are represented in a suffix tree, which is used to identify tandem repeat types and later to identify atomic substrings. For completeness, we briefly review this data structure.

4.1 Suffix trees

A suffix tree (see [9]) is a trie that stores one or more strings and their suffixes. For example, a suffix tree containing the word "banana" also contains "anana" and "nana" and so on. This is shown in Figure 1, where *$* has been added as an end-of-string symbol ensuring that each suffix ends at a leaf node. Interior nodes having only one child are not usually represented as explicit nodes in practice; omitting them (and labeling the edges with strings instead of characters) allows many operations on the data structure to be performed more quickly. This leads to a *compact suffix tree*, illustrated in Figure 2 for the banana example.

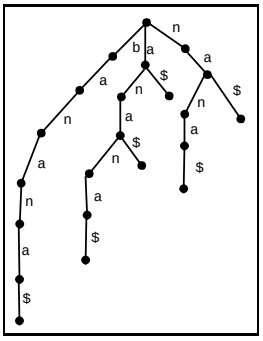

Figure 1. A suffix tree for the string "banana". There is one symbol per edge and this structure is sometimes known as an *explicit suffix tree*.

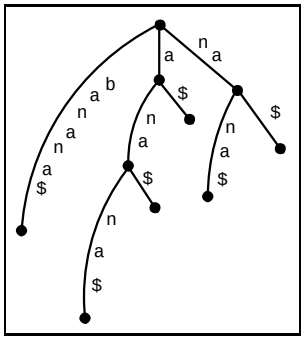

Figure 2. A compact suffix tree for "banana;" nodes with just one child have been removed and the edges are now labeled by substrings rather than symbols.

We use $\overline{v}$ to denote the node in the explicit suffix tree that is reached by starting at the root and traversing the edges labeled by the symbols of v in order. For example, the rightmost leaf node in Figure 1 is denoted by $\overline{\texttt{na\$}}$. If $\overline{v}$ is also a node in the compact suffix tree, we refer to it as an *explicit node*, otherwise it is an *implicit node* in the compact suffix tree.

Suffix trees were previously suggested for intrusion detection by [19]. The idea there was to represent normal program behavior using a suffix tree rather than, say, a database of n-grams as in *stide*. If ASCII characters are replaced by symbols representing system calls, then it is clear that every n-gram in an audit trace s is represented somewhere in a full suffix tree for s. In fact, this is still true if the tree is pruned at depth n, and the resulting data structure can be more compact than an n-gram database.

Our own goal, however, is preprocess audit data *before* it is seen by the intrusion detector, using suffix trees to analyze the structure of the data. As a simple example, one could argue that the labels of the edges of the *compact* suffix tree in Figure 2 represent, in some sense, the vocabulary of substrings that can be found in the word *banana*. We can take advantage of that fact in the construction of intrusion detectors.

The advantage of using suffix trees for such analyses is that many other useful operations can also be performed efficiently. The tree itself can be constructed in linear time [20, 26, 27], and operations like pattern-matching can be performed efficiently once the tree is constructed. One operation of particular interest in this paper is the detection of tandem arrays [10, 24].

4.2 Analyzing the structure of audit traces

It is straightforward to store two or more symbol-strings in a single suffix tree. If this is done with a compact suffix tree, we can then use it to enumerate atomic substrings. Because of the way a compact suffix tree is constructed, the labels of the edges originating at the root are the atomic substrings with respect to strings represented in the tree.

Lemma 1: *If a compact suffix tree is constructed from a set of strings S, then the labels of the edges leading out of the root node and their prefixes are exactly the atomic substrings of S.*

Proof: To show that each substring labeling an edge out of the root node is an atomic substring, we assume the contrary: that such an edge label w has the form uv (meaning that some $s \in S$ has uv as a substring), and that the string uv' also appears in some string $s \in S$, where $v \neq v'$. Without loss of generality, we assume that v and v' differ in their leftmost symbols (if the other conditions hold we can always find a u, v and v' that also cause this condition to be satisfied). S must contain some string that has a suffix starting with uv and some string that has a suffix starting with uv'; this is just a restatement of the assumption that uv and uv' are both substrings of strings that occur in S. Thus, $\overline{uv}$ and $\overline{uv'}$ are nodes in the explicit suffix tree, as is $\overline{u}$. But $\overline{u}$ has two children: one labeled with the leftmost symbol in v, and one labeled with the leftmost symbol of v'. Thus, $\overline{u}$ is an explicit node in the compact suffix tree. But $\overline{w} = \overline{uv}$ is a descendant of u in the explicit suffix tree (since it is a trie on the symbols in these strings), so $\overline{w}$ is not a descendant of the root in the compact suffix tree, which contradicts the original assumption. The same argument shows that each prefix of w is atomic.

On the other hand, if $\overline{w}$ is atomic then no ancestor of $\overline{w}$ has more than one child in the explicit suffix tree. Hence $\overline{w}$ is an explicit node in the compact suffix tree that is a descendant of the root, or else it is an implicit node on an edge leading out of the root. In the former case, w is the label of an edge leading from the root, and in the latter case w is a prefix of such a label. □

The potential usefulness of atomic substrings lies in the fact that they help us model the data to some extent. That is, if w is an atomic substring and we see some prefix of w, then we expect the remainder of w to follow. If we see something else instead, we can regard it as a potential anomaly. On the other had, once we have established that the whole atomic substring w really is there, we can take the position that this substring is normal, and not in need of further analysis. Thus, we can replace it with a meta-token before sending the audit trace on to another intrusion detection algorithm such as *stide*.

For the purposes of intrusion detection, it is also useful to detect tandem repeats. When the audit data are sequences of system calls generated by an executing program, we can make the heuristic assumption that tandem repeats are caused by loops in the program. This lets us represent loops more compactly in the behavior model; for example, we might replace each tandem repeat with a meta-symbol indicating that the corresponding sequence of symbols was seen repeated one or more times. Thus, banana might be represented as bAAa, with the meta-symbol A representing the substring an, or baBB with B representing na.

Of course, it may be that the *number* of times a substring repeats is important for intrusion detection. For example, a loop that is executed too few or too many times may indicate malicious activity. But from a practical standpoint, we cannot represent such information explicitly by keeping a database where the substring is repeated once in one example, twice in another example, three times in the next example and so on potentially ad infinitum. Therefore, even if we decide to store information about the number of times an audit subsequence can be repeated, we have to start by identifying the repeated subsequences themselves.

4.2.1 Detection of tandem repeats

An algorithm for linear-time detection of tandem repeats is given in [10], and requires other algorithms described in [9]. (Demonstration source-code at [11] is also helpful in implementing these algorithms, since the description in [10] is incomplete.) A description of this algorithm would require more space than is available in this paper, but to bring across the general idea we present a simpler (though more expensive) algorithm for tandem repeat detection.

To simplify the exposition we limit ourselves to suffix trees containing only one string. The extension to trees with more than one string is straightforward.

The first step in detecting tandem arrays is to outfit each leaf with a *suffix position*. Recall that each leaf represents the end of some suffix of the string s stored in the tree. The suffix position stored at a leaf is the *beginning* of the suffix represented at that leaf (that is, if w is a suffix of s, then the suffix position of $\overline{w}$ is the position where w starts within s). For example, the rightmost leaf in Figure 2 represents the suffix na, which begins at position 4 in the string banana, hence the suffix position of that leaf is 4.

The second step is to propagate the suffix positions upward to the interior nodes. This is done by giving each interior node a *leaf-list* containing the suffix positions of its offspring. We also compute the *string depth* of each interior node $\overline{v}$, defined to be the number of symbols in the substring v, which is the number of on the edge-labels that must be traversed to reach $\overline{v}$ starting from the root. For example, the edge leading from the root of Figure 2 and labeled na leads

to a node whose string depth is two since the edge-label contains two characters. If we traverse the edge labeled na from *that* node we arrive at a node whose string-depth is 4.

The leaf-lists and string-depths contain enough information to identify the tandem repeats in w.

Lemma 2: *[24] and elsewhere: Consider two positions i and j of some string s, $1 \leq i \leq j \leq |s|$, and let ℓ denote $j - i$. Then the following statements are equivalent:*

- *There is a tandem repeat of length 2ℓ starting at position i in s*
- *i and j both appear in the leaf-list of some node in the suffix tree for s whose string-depth is greater than or equal to ℓ.*

The reader can verify that the lemma holds in Figures 1 and 2. Verifying the general case may also be a helpful exercise for understanding the nature of suffix trees.

Thus, a straightforward algorithm for finding the tandem repeats in a suffix tree is to visit each node and enumerate all pairs of entries on the leaf list that satisfy lemma 2. The enumeration process that takes place at each node is what prevents this from being accomplished in linear time.

Even though our actual implementation *does* use linear-time detection of tandem repeats, additional time and space can still be saved by preprocessing repeats of length one. This can be done without a suffix tree using straightforward algorithm shown in Figure 3. The advantage is that, at least for the data in our evaluations, considerable data compaction can be accomplished by replacing each such repeat with a meta-symbol or with a single occurrence of the repeated symbol. This is useful for two reasons. First, the "linear" in the linear-time algorithm includes the number of repeats that must be found; second, virtual memory systems perform poorly with suffix trees due to the non-local way nodes are accessed, meaning that the suffix tree should ideally fit into real memory. Preprocessing serves to reduce the size of the suffix tree, and it also removes the potentially numerous tandem arrays the are just repetitions of a single symbol.

4.2.2 Stage 1

The algorithm of [10] marks each string with the start- and endpoints of each tandem repeat. This makes it easy to construct a nondeterministic finite automaton that recognizes the tandem repeat types even though some of them might contain tandem arrays themselves. Conceptually, the idea is to construct an automaton that accepts two or more occurrences of any tandem repeat type seen in the training data. Furthermore, if a tandem repeat type contains tandem arrays itself, the automaton must accept them even if those tandem arrays contain a different number of repeats. That is, if u is a tandem repeat type, then, for all w_1, v, w_2 such that u can be written as $w_1 v^n w_2$, the automaton for u must also accept two or more occurrences of $w_1 v^m w_2$ for any $m \geq 1$.

One way to construct such automata is to build a list of tandem repeats sorted in increasing order of length. The shortest repeats on the list are then replaced with meta-symbols in situu (recall that they are represented by their start- and endpoints in the original audit traces) at the same time that automata are constructed to recognize them. This process continues for longer repeats, and as each repeat is processed we scan it for meta-symbols inserted during previous iterations of the procedure. When a meta-symbol for the repeat-type v encountered during the processing of a repeat u, we can insert the automaton that recognizes v into the one that recognizes u. (In other words, if $u = w_1 A w_2$,

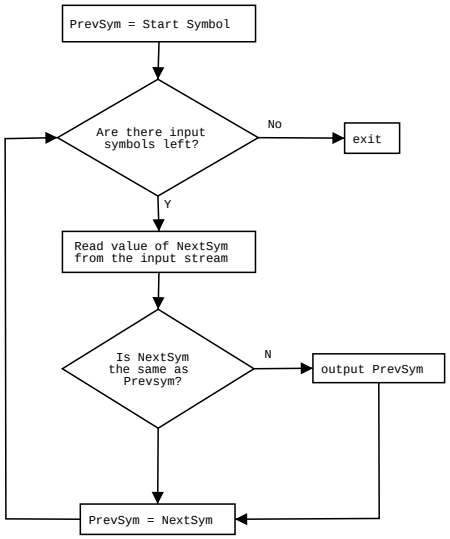

Figure 3. A simple algorithm in which two or more consecutive occurrences of any symbol are replaced by just one occurrence of that symbol. Any symbol that is identical to the previously-seen symbol is simply dropped.

where A a meta-symbol representing the tandem repeat type v, then we start by constructing an automaton that accepts

$$w_1 A w_2 (w_1 A w_2)+,$$

treating A as a symbol. Then we can easily replace A with an automaton that accepts $vv+$.)

Note that each symbol from the original audit stream is scanned only once, and in addition each meta-symbol is scanned only once. But in practice, it is also useful to eliminate redundant tandem repeat types. This is more complicated than it may appear to be at first, because two distinct repeat types that contain embedded repeats may become identical after the embedded repeats are replaced by meta-symbols. For example, the repeat types `baac` and `baaac` are not identical and hence not redundant to one another, but if the repeats `aa` and `aaa` are both replaced by the meta-symbol `A`, then `baaac` and `baac` are both represented as `bAc`, and we we only need to submit one of the two copies of `bAc` to further processing. To eliminate such redundancies, we scan the list of tandem array types many times, resulting an algorithm of superlinear complexity. Nonetheless, the combined lengths of the tandem repeat types are so much smaller *in practice* then the combined lengths of the original audit streams that this elimination of redundancy leads to a performance improvement.

One further subtlety is that, even if we believe that tandem arrays represent loops in the underlying application, the process described above for detecting tandem repeat types may not correctly identify the beginnings and ends of these loops. For example, the sequence of audit events `write write read write write read write` may represent two iterations of a loop that performs `write write read`, followed by a `write` that is not inside the loop. It may also represent two iterations of a loop containing `write read write` preceded by a `write` outside the loop.

We deal with this problem by making each finite state machine accept *rotations* of the repeat type that it was built for. (If α is a symbol and w is a string, then αw is a rotation of $w\alpha$, and all rotations of αw are also rotations of $w\alpha$.)

A single nondeterministic automaton is built for each application by combining the automata that accept the application's tandem repeat types (those smaller automata are also nondeterministic at this point). Figure 4 shows the resulting NDFA, constructed from one week of audit data for the Unix program `pwd` taken from data collected for the 1998 Lincoln Labs intrusion detection evaluations.

Finally, to facilitate faster operation during the detection phase, an equivalent DFA is constructed for each of the NDFAs (one for each application whose audit data will be monitored). This is done using well-known techniques (see [2] for example). While this can be computationally expensive in principle, the simple and regular structure of NDFAs accepting tandem repeat types makes the process reasonably fast. Figure 5 shows the DFA obtained from the NDFA of Figure 4.

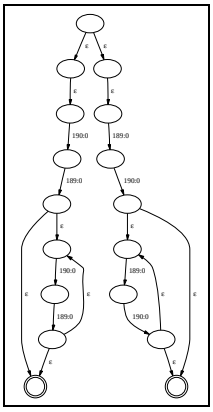

Figure 4. A nondeterministic finite automaton that accepts the tandem repeat types found traces for the application `pwd` in one week of training data from the 1998 Lincoln Labs intrusion detection evaluations. The transitions are labeled with ϵ or numbers representing audit events. In fact, there is only one repeat type consisting of two audit events; the automaton accepts one or more occurrences of this repeat *and* one or more occurrences of its single rotation.

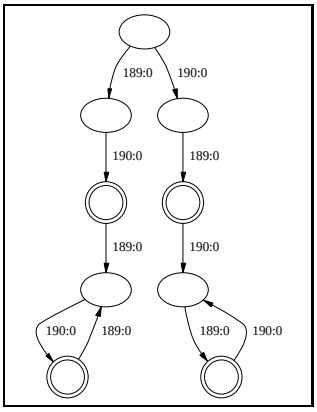

Figure 5. The deterministic finite automaton that accepts the same repeat types.

4.3 Tokenizing audit data

Once DFAs have been constructed to recognize the atomic substrings and tandem arrays in an audit trace, they can be used to tokenize the data. This process is similar to lexical analysis, except that a lexical analyzer reduces the *shortest* substring matching a given regular expression to a

corresponding token, while we reduce the *longest* substring.

One approach is to replace each tandem array and each atomic substring with a meta-symbol unique to it. An alternative is to replace each tandem array with a single copy of the repeated substring. The second approach is somewhat preferable; the first tends to increase the number of unique features in an audit trace even as it decreases the length of the traces. Increasing the number of features would contradict our original intention of simplifying the audit data, and, indeed, it seems to make the task of learning normal application behavior slightly more difficult. Replacing each tandem array with one copy of the corresponding tandem repeat (e.g., the repeated substring) leads to better results.

During training, tokenization is first performed on the audit traces that will be used for training. This is done by using the DFAs that were originally constructed from the same audit traces. These tokenized audit traces are then passed to a learning algorithm (such as *stide*) that then builds an actual anomaly detector. During detection, the same DFAs are used to preprocess new audit traces before they are passed to this detector. The process is illustrated in Figure 6.

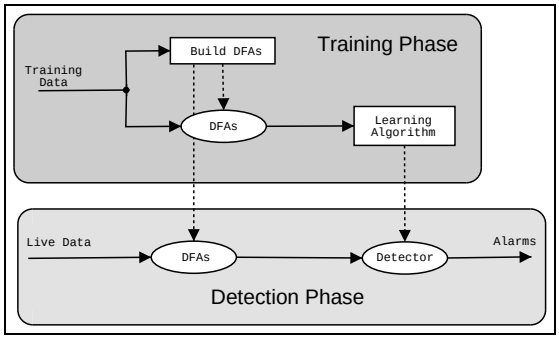

Figure 6. A diagram showing the use of vocabulary extraction in an intrusion detection system. Ovals (the DFAs and the detector) represent artifacts of the training process, and the solid lines represent the fbw of data from audit traces. In the training phase, audit data is used to build DFAs that tokenize the audit data and to build detectors; when the detector is fi elded, the same DFAs preprocess new audit data before it is seen by the detectors.

5 Applications of Vocabulary Extraction

One of the benefits of vocabulary extraction is that can provide significant reductions in data volume. This can be useful simply from the implementation standpoint, since execution traces can contain hundreds of thousands of symbols even for programs that do not execute indefinitely. Tokenization not only reduces the length of execution traces; it can also reduce the number of unique traces, which makes the audit information easier to store. Furthermore, reducing the data volume can make audit data can be easier to process in real time, though this benefit can only be realized if the process of tokenization is efficient itself.

The table below shows the effects of tokenization on audit traces for the Unix program `ps`, taken from the a corpus of data from MIT Lincoln Labs [1]. This data was used for two evaluations of intrusion detection systems, and evaluations took place in 1998 and 1999. The 1998 contained data representing eight weeks of activity on a simulated computer network, and this is the data we use in the table. We show the results for `ps` because the traces were particulary long and numerous, in fairness we should mention that the execution traces from some other applications were compressed by as little as 43%.

Week	Number of unique traces (unprocessed)	Number of audit events (unprocessed)
2	81	14,007
3	150	31,276
4	167	39,868
5	202	66,746
6	99	22,350
7	88	19,355
8	144	30,737
2-8	483	125,872

Week	Number of unique traces (processed)	Number of audit events (processed)
2	24	877
3	34	1,528
4	32	1,942
5	58	3,501
6	25	1,196
7	25	1,375
8	51	1,986
2-8	161	8,750

The table shows the data reduction that results from preprocessing repeats in all eight weeks of the 1998 Lincoln Labs data for `ps`. We see that on average over the 8 weeks of data, the number of unique traces is reduced by nearly 75%, while the number of events is reduced by about 94%. (When all seven weeks are taken together, the number of

unique strings is not the sum of the unique strings for each week, since there is some overlap. Hence the results obtained by processining all seven weeks at the same time are also not sums of the corresponding results from the inidividual weeks.)

5.1 Tokenization in intrusion detection

The main reason for for performing vocabulary extraction is that we hope it will improve the performance of intrusion detectors. We evaluated three techniques for preprocessing audit data before sending it to the intrusion detector.

1. **Simple tokenization:** Each tandem array (sequence of repeated substrings) is replaced by a single metatoken representing the particular sequence of symbols that is repeated. Likewise, sequences of symbols that always occur together are replaced by a metasymbol representing that sequence.

2. **Deflation:** Each tandem array is replaced by just a single occurence of the repeated substring. The actual audit symbols, however, are the same ones that appear in the original trace (they are not metasymbols).

3. **Alarm censoring:** The data is sent to the intusion detector in its original form (or it is deflated), but if an anomaly that occurs entirely within a vocabulary string, that anomaly does not raise and alarm. (Since the regular expression parsers recognize features that occurred in the training data, and since that data is assumed to represent normal behavior, we assume that those features are themselves representative of normal behavior. Therefore an anomaly within a vocabulary string is assumed to be a false alarm.)

6 Evaluation

We applied vocabulary extraction in conjunction with *stide*. The latter detection technique is generally combined with post-processing that averages the last k anomaly scores (for some k), leading to a filtered anomaly score that takes on a value between 0 and 1, depending on how many alarms were raised by the last k audit events. This allows us to set an *alarm threshold*; we signal a possible intrusion only when the filtered anomaly score exceeds this threshold. In our experiments, we applied the same type of post-processing.

Our training data came from several sources, the most important of which was a series simulations of a computer network conducted by Lincoln Laboratories in 1998 and 1999. We have twelve weeks worth of this data, but at the time these experiments were conducted we had not yet separated intrusions from benign behavior in one week of the data. Some further data was supplied by Johns Hopkins University. Finally we discovered that we needed additional data describing the normal behavior of five programs: `sendmail`, `eject`, `in.ftpd`, `fdformat`, and `xterm`, so we collected data for this programs on our own system and by using automatic test data generation techniques such as those described in [21].

The data contains evidence of several kinds of attacks, which can be divided into two broad classes: (1) probes and denial-of-service attacks, and (2) unauthorized accesses and unauthorized privilege elevations. Our systems are meant to detect the second class of attacks, so we did not evaluate them on attacks in the first class.

Furthermore, some attacks do not leave identifiable traces in the audit data. For example, some attacks consist of moving files to a location where they should not be. Identifying such attacks means knowing what locations are disallowed; in other words, the intrusion detector must know the details of the system's local security policy. Our prototypes currently have no knowledge of local security policies, so they do not detect such attacks. Another attack involves setting up a malicious HTTP client to transfer information off of the system using cookies. This attack does not involve misuse of any existing programs, so (strictly speaking) our systems cannot detect such an attack either. In reality, such attacks are sometimes detected because of statistical variations between the behavior of benign users and malicious users, but we do not know how easily an intruder could avoid this sort of detection. Moreover, our systems are meant to detect program misbehavior, not profile computer users. Therefore we decided that these particular attacks are out of the intended scope of the system. As a result, just under 93% of the access and elevation-of-privilege attacks are in the scope of our system.

There are 181 types of system calls recorded in the audit data, and these events form the inputs to our intrusion detection system. The events are collated into sequences of audit events generated by individual programs, and a different intrusion detector is trained for each program. The anomaly score for a session is the maximum of the anomaly scores

for the programs in that session

The experimental data is divided into user sessions, some of which contain intrusions. Our intrusion detectors examine all program executions that occur in a session, and the highest anomaly score for any of these programs is used as the anomaly score for the session.

To quantify the performance of a given system on a given set of data, we measure how many benign sessions are falsely marked as being intrusive, and measuring how many intrusive sessions are overlooked. All three systems output a number between 0 and 1 describing how anomalous a given session is, so the performance of any given system depends on the threshold at which we raise the alarm. By varying the threshold and plotting the percentage of false alarms against the percentage of missed intrusions, we obtain a plot similar to a receiver operating curve, which is a convenient way to visualize the performance of the intrusion detectors.

We tested our approach using seven-fold cross validation. In a series of seven experiments, the system was trained using all of the data except for one of the seven weeks of data we had from Lincoln Laboratories. The system was then tested using the remaining week of data, in order to test immunity to false alarms. None of the audit data reflecting intrusive activity was used during training, so the systems were tested on all of the intrusive data during each of the seven cross-validation phases.

The *stide* algorithm was tested with a number of different choices for the n-gram length. In the work of [7] and elsewhere, it was reported that six was the best choice for n, but this is not so for the Lincoln Labs data, where the best choice for n appears to be three (see [14, 25] for a discussion of why different choices work in different settings). For each value of n between three and ten, we tested the algorithm with and without vocabulary extraction.

Figure 7 shows curves comparing the performance of the detectors with and without alarm censoring.

The best performance for both detectors was at n=3; this value of n led to the lowest false alarm rates while still allowing detection of all the attacks that are within the capabilities of the detectors. At this level, the false alarm rate was reduced from 0.47% to 0.38%, about a twenty percent improvement.

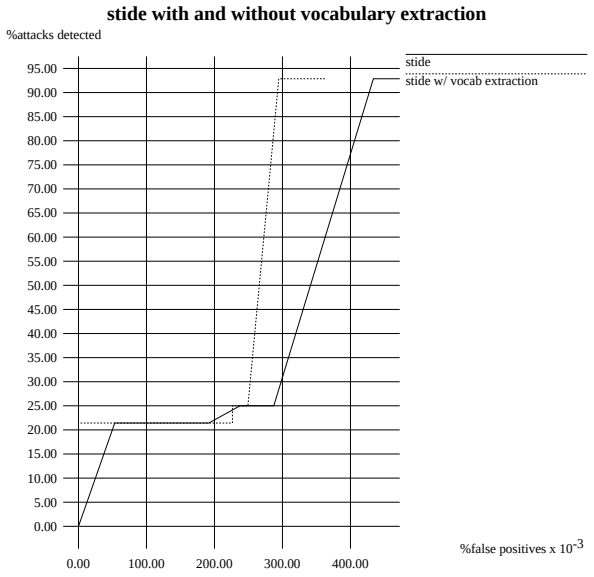

Figure 7. The performance of 3-stide with and without vocabulary extraction. The vertical axis represents the percentage of attacks detected, and the horizontal axis represents the percentage of user sessions that raised false alarms.

7 Resource Requirements

Our suffix trees are constructed using the Ukkonen algorithm (see [9]) whose time and space requirements grow linearly in the number of symbols that are inserted into the suffix tree. The algorithm for detecting tandem repeats was also linear in complexity. However, one notorious shortcoming of suffix trees is that they perform poorly when they have to be stored in virtual memory. Elements of a suffix tree have to be accessed in a non-localized way, so the caching and pre-fetching heuristics of virtual memory managers tend to be ineffective. Hence suffix-tree algorithms tend to cause excessive swapping once the trees become to large for real memory.

Another potential performance issue is the construction of regular-expression parsers. Efficiency is of great importance during detection, so the parsers should be deterministic, but the encoding of a set of tandem repeats as a regular expression generally leads to a nondeterministic parser. We used a standard algorithm to convert the nondeterministic parsers to deterministic ones (see [2] for example), but this algorithm can also be time-consuming.

In practice, we found that our suffix trees did not grow

into virtual memory. Our evaluations were conducted on machines with 256M of real memory, but we did not optimize the space requirement of the trees (indeed, our architecture made in convenient for implementation reasons to store each string twice in the suffix tree structure, once verbatim and once in the tree itself). However, the detection of atomic substrings requires that all audit traces for an application be stored in a single suffix tree, and this proved impractical unless the audit traces had already been compressed by processing their tandem arrays. In other words, practical performance considerations prevent the application of atomic substring in isolation, though tandem array detection can be applied by alone.

It appears that our execution traces were well-suited for the application of suffix trees. Since there are 181 types of symbols in our audit traces, each node in the suffix tree could have as many as 181 children, but in practice the average number of children per node appears to be in the low single digits due to the relatively regular behavior patterns of the programs being monitored.

The conversion of nondeterministic regular expression parsers into deterministic ones also was no hurdle in practice. Once again, the regular structure of the execution traces seems to come to our aid; the set of states reachable by non-deterministic transitions from a given state also appears to be comparatively small on average.

At the time of this writing, we do not exact timing information, but the end-to-end processing of our training data (which contains some twenty-five million events in 169,251 execution traces from thirty-seven applications) takes about twenty-five minutes on a 600MHz Pentium processor.

8 Conclusion

This paper described simple methods for abstracting irrelevant details from the execution traces used by program-based anomaly detectors. An empirical evaluation showed a twenty-percent reduction in the false-alarm rate when this technique was applied to the well-known *stide* algorithm. The strength of this approach, however, is not its performance in conjunction with *stide*, but the fact that it can be applied to a large set of program-based anomaly detectors. Preliminary results with several other detectors suggest a comparable (or better) reduction in false alarm rates. Vocabulary extraction can also be applied in conjunction with different techniques for monitoring application behavior, such as approach of [23], which in itself provides audit data at a higher level of abstraction than the BSM subsystem we used in our evaluation.

Some interesting open problems remain. One especially pernicious source of false alarms — one that does not appear in the Lincoln Labs data because of the nature of the Unix programs used there — is the presence of asynchronous events. In many applications, certain behaviors may occur at any time, triggered by user actions, sensors, or communication among threads, to give just three examples. It would be beneficial to identify these asynchronous behaviors and either censor them or process them in isolation from the rest of the behavior data. This seems to be a natural application of vocabulary extraction, and one that we have not yet explored extensively.

References

[1] Mit lincoln labs. 1999 darpa intrusion detection evaluation, `http://www.ll.mit.edu/ist/ideval/index.html`.

[2] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[3] J. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, April 1980.

[4] J. Cannady. Artifi cial neural networks for misuse detection. In *Proceedings of the 1998 National Information Systems Security Conf erence (NISSC'98)*, pages 443–456, October 5-8 1998. Arlington, VA.

[5] P. D'haeseleer, S. Forrest, and P. Helman. An immunological approach to change detection: Algorithms, analysis an d implications. In *IEEE Symposium on Security and Privacy*, 1996.

[6] D. Endler. Intrusion detection: Applying machine learning to solaris audit data. In *Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98)*, pages 268–279, Los Alamitos, CA, December 1998. IEEE Computer Society, IEEE Computer Society Press. Scottsdale, AZ.

[7] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society, IEEE Computer Society Press, May 1996.

[8] A. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 1998 Annual Computer Security Applications Conf erence (ACSAC'98)*, December 1998.

[9] D. Gusfi eld. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Cambridge, Mass., 1997.

[10] D. Gusfi eld and J. Stoye. Linear time algorithms for fi nding and representing all the tandem repeats in a string, 1998.

[11] J. Knight, D. Gusfi eld, and J. Stoye. The strmat software-package, `http://www.cs.ucdavis.edu/ gusfield/strmat.tar.gz`, 1998.

[12] T. Lane and C. Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conf erence*, pages 366–377, October 1997.

[13] W. Lee et al. Learning patterns from Unix process execution traces for intrusion det ection. In *Proceedings of AAAI97 Workshop on AI Methods in Fraud and Risk Man agement*, 1997.

[14] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[15] T. Lunt. Ides: an intelligent system for detecting intruders. In *Proceedings of the Symposium: Computer Security, Threat and Count ermeasures*, November 1990. Rome, Italy.

[16] T. Lunt. A survey of intrusion detection techniques. *Computers and Security*, 12:405–418, 1993.

[17] T. Lunt and R. Jagannathan. A prototype real-time intrusion-detection system. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.

[18] T. Lunt, A. Tamaru, F. Gilham, R. Jagannthan, C. Jalali, H. Javitz, A. Valdos, P. Neumann, and T. Garvey. A real-time intrusion-detection expert system (ides). Technical Report, Computer Science Laboratory, SRI Internationnal, February 1992.

[19] C. Marceau. Characterizing the behavior of a program using multiple-length N-grams. In *Proceedings of the 2000 new security paradigm workshop*, pages 101–110, 2000.

[20] E. M. McCreight. A space-economic suffi x tree construction algorithm. *Jrnl. A.C.M.*, 23(2):262–272, Apr. 1976.

[21] C. C. Michael and G. E. McGraw. Generating software test data by evolution. *IEEE TSE*, 27(9):1085–1110, 2001.

[22] P. Porras and P. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, October 1997.

[23] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 144–155. IEEE Computer Society, 2000.

[24] J. Stoye and D. Gusfi eld. Simple and flexible detection of contiguous repeats using a suffi x tree. In *Proc. 9th Annual Conference on Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Computer Science*, pages 140–152. Springer-verlag, 1998.

[25] K. M. C. Tan and R. A. Maxion. Why 6? defi ning the operational limits of stide, an anomaly-based intrusion detector. In *IEEE Symposium on Security and Privacy*, pages 12–15, 2002.

[26] E. Ukkonen. Constructing suffi x trees on-line in linear time. In J. van Leeuwen, editor, *Proceedings of the IFIP 12th World Computer Congress. Volume 1: Algorithms, Software, Architecture*, pages 484–492, Amsterdam, The Netherlands, Sept. 1992. Elsevier Science Publishers.

[27] P. Weiner. Linear pattern matching algorithms. In *Conference Record, IEEE* 14^{th} *Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.