Characterizing the Behavior of a Program Using Multiple-Length N-grams

Carla Marceau
Odyssey Research Associates[1]
Ithaca, NY 14850
carla@oracorp.com

ABSTRACT

Some recent advances in intrusion detection are based on detecting anomalies in program behavior, as characterized by the sequence of kernel calls the program makes. Specifically, traces of kernel calls are collected during a training period. The substrings of fixed length N (for some N) of those traces are called N-grams. The set of N-grams occurring during normal execution has been found to discriminate effectively between normal behavior of a program and the behavior of the program under attack. The N-gram characterization, while effective, requires the user to choose a suitable value for N. This paper presents an alternative characterization, as a finite state machine whose states represent predictive sequences of different lengths. An algorithm is presented to construct the finite state machine from training data, based on traditional string-processing data structures but employing some novel techniques.

Keywords

Intrusion detection, computational immunology, finite automata, string processing.

1. INTRODUCTION

In [5, 6, 8], Stephanie Forrest has shown that the behavior of a program can be characterized by the sequence of calls it makes to the operating system kernel. Forrest and her colleagues at the University of New Mexico have used this observation to develop a novel and effective method of intrusion detection. During a *training* period, system calls from the process running the program are collected and characteristic patterns of system calls are placed in a database. Once the normal behavior of the program has been fully characterized in this way, the database can be used for intrusion detection. To detect intrusions, system calls are again collected, but this time they are compared against the contents of the database. When clusters of discrepancies between the run-time behavior and the database are found, it is likely that an attack has occurred. This method of intrusion detection has been used to detect attacks in which some other program takes over a process (for example, in a buffer overflow attack) or in which a program is being used in an illegitimate way. The method is simple and effective, can be tuned to practically eliminate false alarms, and can detect novel attacks because it is based solely on a program's normal behavior, not on characteristics of any particular attack.

This approach to intrusion detection is part of a program of research called computational immunology, whose goal is to build a computer immune system, by analogy with the vertebrate immune system. Intrinsic to the vertebrate immune system is the distinction between "self"—the individual organism—and other, possibly pathogens. Analogously, the database that characterizes normal data is called the *self database*, because it describes the "self" of a program, or its normal execution behavior.

The key to the success of this approach is the method used to characterize normal program behavior. A *trace* is a sequence of observations of the program's behavior, for example a sequence of calls made to the operating system kernel by the process executing the program. Any non-trivial program has an infinite number of potential traces, which must somehow be characterized with a finite database. An essential insight was that the database could consist of short sub-strings that may occur in a trace. Specifically, consider each sub-string of length N (commonly called *N-gram*) of a trace. The set of all distinct N-grams that occur in all normal traces during the training period is considered to empirically characterize the program's normal behavior. For sufficiently large N, that set is only a small subset of all possible N-grams, hence the remarkable effectiveness of the computational immunology approach.

Forrest's simple and effective method has been used to detect attacks on Unix privileged processes. To make it work with a new operating system, one needs to instrument the operating system and to choose a suitable value for N. The appropriate value for N depends on the granularity of the kernel calls, which may vary from one operating system to another.

We have applied this method to distributed applications [10]. In our distributed environment, it was natural to use traces of calls from client to server, rather than the more traditional kernel calls. Using application server calls as elements of the traces meant that we had to find a value for N that was suitable for a given application. This was a significant problem, since the granularity of the calls can vary widely from one application to another. Further, in an operating system environment, one can assume that operating system experts are able to instrument the system and find some appropriate value for N. In a

New Security Paradigm Workshop 9/00 Ballycotton, Co. Cork, Ireland
ACM ISBN 1-58113-260-3/01/0002

[1] This work was sponsored in part by the Defense Advanced Research Projects Agency of the Department of Defense, monitored by the Air Force Research Laboratory under Contract F30602-97-C-0126.

distributed application environment, the experts on server operations are the application developers. We discovered that the task of finding a suitable N is hard to explain to application developers and easy to get wrong, so we resolved to find a way to help the developers perform it.

The solution turned out to be not to choose a value for N, but to automatically find a set of strings of different lengths (multiple-length N-grams) that describes normal execution. This turned out to have several advantages, including simplifying the detector and possibly reducing the length of time required to find a characteristic set of strings.

In the remaining sections of this article, we first briefly review Forrest's method of characterizing programs and the training cycle needed to characterize normal program behavior. We then show how to use traditional string processing techniques to find a set of strings of different lengths that is equivalent to the N-gram characterization. Finally, by increasing N but relaxing the requirement for an exact match, we arrive at a new characterization that finds a set of strings of "appropriate" length, without sacrificing the ability to detect anomalies.

2. The N-gram Characterization of Program Behavior

It is important to note that the self database for a program consists of *all* N-grams that occur during training. That is, we do not slice each trace into segments of length N. Instead, we collect the contents of a window of width N as it slides along the trace—hence Forrest's detection algorithm is called the *sliding window* algorithm. For example, consider a trace of training data, where we use letter symbols to stand for kernel calls:[2]

```
A B C C A B B C C A A B C
```

If N = 4, the self database consists of the set of all substrings of length four of the training string:

```
ABCC
BCCA
CCAB
CABB
ABBC
BBCC
CCAA
CAAB
AABC
```

After the database has been constructed, we can check executions of the program against the self database. An *anomaly* is an N-gram that occurs during execution but is not in the database. For example, the string ABCCABBCCABABC contains the anomalies CABA, ABAB, and BABC. A few anomalies may simply mean that the training traces did not cover all normal behavior, but a large cluster of anomalies is a good indicator of an attack.

[2] Note that this string is too short to be considered adequate for training for even a toy program. It is presented here merely to illustrate the concepts involved.

Two factors influence the length of time needed for training. First, it is important to continue training until almost all "normal" behaviors have occurred, because otherwise there is a real risk of a high false alarm rate. Complex applications may have many behaviors and require a long training period. Second, the choice of N affects the training time, because the set of (N+1)-grams is naturally larger than the set of N-grams, for any given trace. In this paper, we will be concerned with the second factor, which implies that the value of N should not be too large. On the other hand, a value of N that is too small can result in a poor characterization of normal behavior. Choosing N = 1, for example, simply characterizes a program by the set of kernel operations it calls, which is almost always inadequate.

It is desirable, then, to use the smallest value of N that can adequately distinguish between normal and abnormal program behavior.

2.1 The training problem—why the value of N varies

In applying the N-gram characterization to distributed applications, we were concerned with characterizing the behavior of an application client as it appears to a server from which it requests service. When the observer of a program is the operating system kernel, the granularity of kernel calls may determine a suitable value for N. Some experimentation with the operating system would then be sufficient to find this value. In the case of distributed applications, however, each server determines the granularity of the calls it accepts. The responsibility for finding a suitable value for N falls on the application developer.

Training is typically done in one of two ways. One way is to make up a set of representative test cases and exercise the program using them. Another is to collect data during actual use of the program. In either case, it is necessary to decide when training is complete—incomplete training will lead to false positives. At the beginning of training, the self database grows very rapidly; later the rate of growth decreases. When the rate of growth becomes very small, we say that the self database *converges*. At this point, it is likely that the self database is nearly complete, and training can stop.

If N is too small, false negatives are likely. On the other hand, increasing the value of N significantly increases the time required for training. Hence, one would like to use the smallest value of N that is adequate for avoiding false negatives—but since training is based only on normal behavior, that value is not known in advance.

It should be obvious that choosing a value for N poses a problem for the application developer. He would prefer to throw the training data into a hopper and obtain either a self database or a message telling him that the self database does not yet converge.

2.2 Representing self with multiple-length N-grams

The sliding window N-gram approach works because small substrings are good predictors of the next symbol to be encountered in a process trace. In principle, it seems likely that the number of symbols needed for predicting the next one

varies from one point in a process to another. Our approach is to look for shortest strings that are good predictors of the next symbol to occur. Consider, for example, the string "ABCABCABCABC." We would characterize this very repetitive string by the substrings "A," "B," and "C," because each symbol accurately predicts the following symbol. A rather different approach that also uses modern text processing techniques is discussed in the Related Work section.

Figure 1 shows a more realistic example of the strings that result from our construction. The boldface string at the top is a short piece taken from Forrest's synthetic training data for `lpr` [2], in which symbols have replaced the original data values. The lines below the boldface string show successive (multiple-length) substrings that "cover" the trace fragment. Each substring (e.g. CEAB or CD) consumes one more symbol of the input than the preceding substring.

```
G A H C E A B C C C D A B C C
G A H
    H C
    H C E
      C E A
      C E A B
          B C
              C
                C
                C D
                C D A
                C D A B
```

Figure 1. Multiple-length strings covering a test string

Intuitively, the strings of Figure 1 are selected for their predictive power. The fact that the current string is CEA, for example, constrains the (legitimate) possibilities for the subsequent symbol. Note that this fragment is too short for the predictive power of the strings to be apparent. It is presented here solely to provide an idea of how the multiple-length strings cover a trace.

3. Construction of a self database with multiple-length strings

In this section, we describe our construction. The description is in three steps:

(1) We construct a suffix tree for N-grams of the training data, for some value of N that is "large enough." N will be the upper limit on the length of the multiple-length N-grams. Note that picking a "large enough" N for variable-length N-grams is much easier than picking a good value for fixed-length N-grams. A finite state machine (FSM) for the two-finger algorithm (equivalent to Forrest's sliding window algorithm) can be constructed immediately from the suffix tree.

(2) We can compact the suffix tree to a directed acyclic graph (DAG) by merging "equivalent" subtrees. The result is a set of strings of varying lengths that is equivalent to the original set of N-grams. The DAG also gives rise to a FSM.

(3) By introducing a variation on the compaction, we can merge two subtrees if they are "almost equivalent." This enables us to introduce a bias toward short strings by removing longer strings that are approximately equivalent to shorter ones. The result is a weakening of the sliding window algorithm, for the large value of N chosen. Intuitively, choosing a large value of N increases the size of the self database by introducing long strings, some of which are arbitrary concatenations of shorter ones. The approximation step removes longer strings that are approximately equivalent to shorter ones.

3.1 The Two-Finger Algorithm

In [6], Forrest defines the self of Unix processes in terms of a sliding window (of constant width N) over the sequence of system calls. "Self" is the set of strings of length N that appear in training traces. In this section, we show how to derive a FSM implementation of the sliding window algorithm.

We begin with a suffix tree, a data structure commonly used in string searching algorithms (see, for example, [7]). A suffix tree for a set of keywords (in our case, the set of N-grams) is a tree each of whose nodes corresponds to a distinct suffix of one or more of the keywords. Figure 2 shows a suffix tree for the 4-grams of the example trace.

In general, a suffix tree for a string contains every suffix of the string—so the suffix tree for a string of length m has m distinct suffixes. The suffix tree for a set of strings contains every suffix of each string. A suffix tree for a set of N-grams can be constructed by first building a suffix tree for the set of training strings and then truncating each branch at depth N.[3]

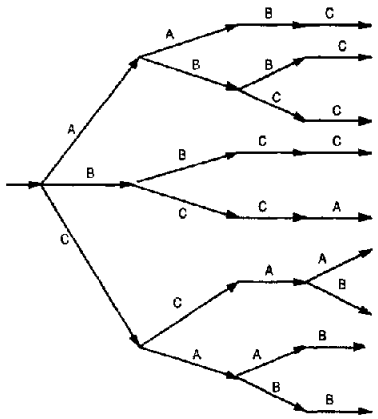

Figure 2. A suffix tree for the sample string A B C C A B B C C A A B C

Each node at depth k ($k \leq N$) in the tree is *labeled* with a k-gram of the training strings. The leaf nodes at depth N are labeled with N-grams (ABCC, ABBC, etc.). The root node is labeled with the empty string.

[3] The suffix tree will in general include branches of depth less than N, representing final suffixes of the training strings. This distinguishes it from a *keyword tree* for the N-grams, which has the same branching structure but each of whose branches contains exactly N edges.

Each edge is labeled with a symbol. For example, an edge labeled A goes from node C to node CA.

Suffix links connect each node to its longest proper suffix. For example, the suffix link of the node labeled with ABCC goes to the node labeled with BCC.[4] Suffix links are not labeled; the suffix link from a node corresponds to all labels that do not occur on out-edges of that node. The root does not have a suffix. Figure 3 depicts the suffix tree of Figure 2 with a few suffix edges shown as dashed lines (inserting all of the suffix edges would make the graph hard to read). For example, the suffix of ABCC is BCC. The suffix of BCCA is CCA. All suffix edges for the path BCCA are included.

The suffix links provide a way to go from an N-gram (a leaf node) to the following N-gram—it is merely necessary to follow the suffix link from the leaf node and then the edge labeled with the last symbol of the following N-gram. For example, from ABCC, one can follow the suffix link to BCC, from which an out-edge leads to BCCA. Suffix links are also useful in handling anomalies, as will be seen below.

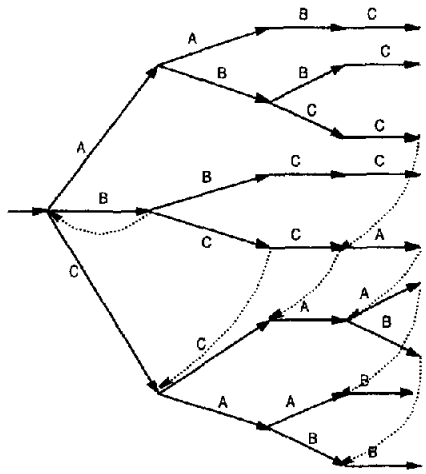

Figure 3. Suffix tree with some suffix pointers

Out-edges from the root node include only the symbols that appear in the training data. New symbols can occur during detection—for example, in this case, if we encounter a "D." Hence, our detection algorithm needs one more node, which is labeled "UNKNOWN_ SYMBOL."

The two-finger detection algorithm can be described as follows. Imagine that while reading a test string, you enclose a substring of letters with your two index fingers. As you read the string, you move your fingers so they always contain the current contents of the sliding window. Suppose you are reading the test string ABCCABBCCABABC. You begin with both fingers together at the beginning of the string. This (initial) state corresponds to the root node of the tree, or the empty string. Moving your right finger one letter to the right corresponds to descending one level (along a labeled edge) in the tree of Figure 3. In this case, with successive moves of your right finger, you successively visit states A, AB, ABC, and ABCC. However, we require that your fingers be separated by at most N letters. In order to move past the Nth letter, you must move *both* fingers one letter to the right. The corresponding action in the suffix tree is to move along the suffix link from the current node to the node representing the suffix of the label of the current node. For example, from state ABCC you move your left finger and arrive at state BCC. (Note that the use of suffix trees ensures that this state is in the tree.) You may now move your right finger again, which brings you to state BCCA.

The right-finger moves correspond to edges in the tree, while left finger moves correspond to the suffix links. We can imagine augmenting the tree with additional "branches" corresponding to the two-finger moves. It is easy to see how, using left-, right-, and two-finger moves, we can traverse any string that contains no anomalies. After reading in an initial N symbols from a completely normal trace, the FSM executes an alternating series of two-finger moves, each of which ends with the fingers enclosing an N-gram.

An anomaly requires us to use left-finger moves. The example test string given above is identical to the training string, except that after the tenth character, an additional B has been inserted, so that the string ends with "CCABABC" instead of "CCAABC." From the node labeled CCAB, we follow a suffix link (left-finger move) to CAB. Since there is no node CABA, we follow the suffix link to AB, but that node also has no out-edge labeled A. We then follow another suffix link to B, and finally a suffix link to the root, which does have an out-edge for A. The three extra suffix links correspond to three anomalous N-grams in the input string (CABA, ABAB, and BABC). From the root, edges descend to ABC.

Based on the suffix tree, it is straightforward to define a finite state machine that implements Forrest's sliding window algorithm. First, we recall the definition of a state-output automaton, which produces output for each state. Then we define a suffix automaton to be a state-output machine that captures the way we use suffix links as "none of the above" transitions. Finally, we will show how the suffix tree gives rise to a suffix automaton.

Definition. A *state-output machine* (following [1]) is a quintuple $(S,\Sigma,\delta,Y,\beta)$, where

(1) S is a finite set of states.

(2) Σ is a nonempty finite set (the input alphabet). Σ^* is the set of strings of input symbols. Similarly, Σ^k is the set of strings of length k.

(3) δ: $S \times \Sigma \rightarrow S$ is a transition function. The function δ can be extended to strings of input symbols in the obvious way.

(4) Y is the set of outputs.

(5) β: $S \rightarrow Y$ is an output function from states.

Notice that the output is determined by the state. Upon entering a state, the FSM emits an output symbol. In this paper, we are concerned only with $Y = \{0,1\}$, where 1 signifies an anomaly and 0 the absence of an anomaly. If $\beta(s) = 0$, we call s an accepting state.

[4] When used in string searching algorithms, suffix trees typically have fewer nodes and a string of symbols per edge. To implement error recovery in the sliding window algorithm, we need a node for each k-gram.

Definition. A *suffix automaton* is an 9-tuple $(S,\Sigma,\Delta,s_0,s_\omega,\varphi,\mu,Y,\beta)$, where S, Σ, Y, and β are the same as for a state-output machine. A suffix automaton has partial functions that define ordinary state transitions for some input symbols (Σ_s) for each state s. For input symbols not in Σ_s, the effect of the input symbol is defined via a suffix function φ.

To define the execution of a suffix automaton, we need Δ, s_0, s_ω, φ, and μ. Δ defines the "normal" transitions, corresponding to the branches of the suffix tree, and φ defines the suffix transitions. We will need s_0, s_ω, and μ to ensure that the transition function for the suffix automaton is well defined.

Δ is a set of partial functions δ_s. For each node s, let δ_s: $\Sigma \rightarrow S$ define ordinary state transitions for some subset Σ_s of Σ. Like δ, δ_s can be extended to strings of input symbols.

The suffix function φ: $S \rightarrow S$ defines the suffix of each node except s_0. We will define execution at s to take the transition δ_s if possible, and otherwise to take a transition defined by $\varphi(s)$. If $\varphi(s)$ doesn't have an appropriate transition, we try $\varphi(\varphi(s))$, and so on. In order to ensure that the transition is well defined, we must make sure that following suffix transitions does not lead into an infinite loop. We require that there exists a function μ: $S \rightarrow \mathbb{N}$, from the states S to the natural numbers, such that $\mu(s) = 0$ iff $s = s_0$ and that for all $s \neq s_0$, $\mu(\varphi(s)) = \mu(s)-1$. This ensures that all members of the sequence $s,\varphi(s),\varphi(\varphi(s)),\ldots$ are distinct and that every such sequence terminates (at s_0). We let $\Phi(s)$ denote the sequence starting at state s.

The special state s_ω is needed in order to define transitions at s_0 if $\Sigma - \Sigma s_0$ is not empty, since s_0 has no suffix state. Σs_ω is empty and $\varphi(s_\omega) = s_0$.

We can now define execution of a suffix automaton by providing the transition function for the equivalent state-output machine. For each state s and each input symbol x,

$\delta(s,x) = \delta_s(x)$ if $x \in \Sigma_s$

$\delta(s,x) = \delta(\varphi(s),x)$ if $x \notin \Sigma_s$ and $s \neq s_0$,

$\delta(s_0,x) = s_\omega$ if $x \notin \Sigma s_0$.

That is, if there is a transition from s labeled x, it defines the following state. If not, and if s has a suffix state, the suffix state defines the following state. However, one state, s_0, has no suffix. If s_0 has no transition labeled x, the following state is s_ω.

Having defined the general operation of a suffix automaton, we now show how to construct a suffix automaton for a given suffix tree.[5] In such an automaton, S contains one state for each node in the suffix tree. Because of the nature of the sliding window algorithm, $\Sigma_s \subseteq \Sigma_{\varphi(s)}$.

The suffix links define φ. The function μ that we need for ordering suffix chains is the depth of s in the tree. The state s_0 corresponds to the root node and s_ω to the UNKNOWN_SYMBOL node.

The branches of the suffix tree specify Δ in the obvious way for nodes with depth less than N. In the example of Figure 3, Σ_{AB} = {B,C} and $\delta_{AB}(C)$ = ABC. For nodes at depth N, we augment Δ with functions corresponding to the two-finger moves. Formally, $\Sigma_s = \Sigma_{\varphi(s)}$ and $\delta_s(x) = \delta_{\varphi(s)}(x)$ for all x in $\Sigma_{\varphi(s)}$. For example, $\delta_{ABCC}(A)$ = BCCA.

The set Y of output symbols is {0,1}. The output function β is simply defined. $\beta(s_\omega) = 1$. For $s \neq s_\omega$, $\beta(s) = 0$ iff s corresponds to a node of the suffix tree at depth N. (Note that the automaton does not emit any output for the first N-1 symbols. The output can be suppressed by filtering the output of the automaton through some counting device.)

3.2 Compressing the finite state machine

By the nature of the sliding window algorithm, the suffix tree exhibits a good deal of redundancy. We now remove the redundancy by identifying and merging equivalent states. The intuitive notion of equivalence we use here is that if two suffixes of an N-gram are equally good at predicting which N-grams can follow (without causing an anomaly), then we will consider them equivalent. In the case of anomalies, this will enable us to move our left finger over several symbols instead of just one symbol at a time. An example of redundancy may be seen in Figure 4, where similar portions of the tree of Figure 2 are shown as dotted lines (note that φ(AB) = B).

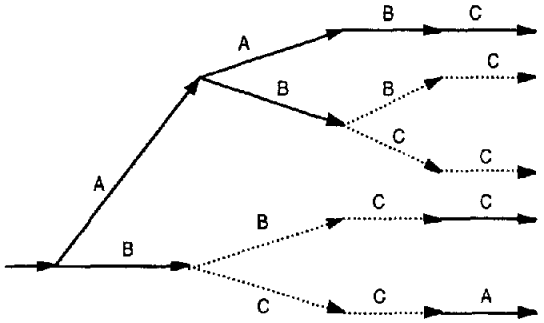

Figure 4. Redundant portions of (part of) the suffix tree

We now formalize this notion of similarity between subtrees as an equivalence relation between states of a suffix automaton that is based on a suffix tree. Recall that $\Phi(s)$ is the sequence of suffix states of s. We define two states to be *similar* if they are members of a substring of $\Phi(s)$ and their normal (branch) transitions go to equivalent states. (For example, in Figure 4, AB and B are similar.)

Definition. If $s_1 = \varphi(s_2)$ or $s_2 = \varphi(s_1)$, states s_1 and s_2 are similar ($s_1 \sim s_2$) iff

(a) $\Sigma_{s_1} = \Sigma_{s_2}$ and

(b) $\forall x \in \Sigma_{s_1}$, $\delta_{s_1}(x) = \delta_{s_2}(x)$ or $\delta_{s_1}(x) \sim \delta_{s_2}(x)$.

This is well defined for a suffix automaton derived from a suffix tree, because for a node s at depth less than N, $\delta_s(x)$ has greater tree depth than s, and the tree depth is bounded by N. Note that by construction, every node s at depth N is similar to its suffix (because $\Sigma_s = \Sigma_{\varphi(s)}$ and $\delta_s(x) = \delta_{\varphi(s)}(x)$ for all x in Σ_s). We let $\approx$ denote the transitive closure of $\sim$. Then $\approx$ is

[5] In this discussion, we will usually not distinguish between nodes of the tree and the corresponding states. The mapping should be obvious.

an equivalence relation on S, and we can form the quotient set $S/\approx$. We let $[s]$ denote the equivalence class of s in $S/\approx$.

The states $S/\approx$ form a suffix automaton. Δ, s_0, s_ω, and Y map to $S/\approx$ in the obvious way, but we must still define the suffix function $\hat{\varphi}$, the depth function $\hat{\mu}$, and the output function $\hat{\beta}$ on members of $S/\approx$. We want the suffix of the equivalence class to be the suffix of its smallest member (e.g., we want the suffix of [AB] to be the equivalence class of the suffix of [B], since AB $\approx$ B). Informally, we consider B to be the representative member of $S/\approx$. Formally, consider the members of $\Phi(s)$— s, $\varphi(s)$, $\varphi(\varphi(s))$, etc. Let $\hat{s}$ be the last member of the sequence that is similar to s, i.e., the member that is highest in the tree. Then we define $\hat{\varphi}([s])$ to be $[\varphi(\hat{s})]$. Similarly, we can define the depth function on $S/\approx$ by $\hat{\mu}([s]) = [\mu(\hat{s})]$.

We define $\hat{\beta}$ (the accepting states) slightly differently in order to ensure that some states of $S/\approx$ are accepting. We define an equivalence class to be accepting if it is the image of *any* accepting state. Consider the set of all states s' that are similar to s, and consider the set of $\mu(s')$. $\hat{\beta}([s]) = 0$ iff N is in the set.

The resulting automaton for the example of Figure 4 is shown in Figure 5 (state names are omitted in Figure 5). All nodes are accepting, except the root and nodes labeled [A], [B], and [C]. Note that states ABBC, BBC, and BC from the original automaton all map to [BC]. The suffix state for [BC] is [C], the image of the suffix state of BC. [BC] is an accepting state, because ABBC is accepting. The accepting states are [AA], [AAB], [BC], [CC], [CA], [CAB], and [BB].

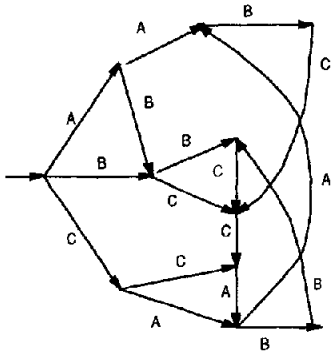

Figure 5. Automaton resulting from compression of Figure 2

Note that $\approx$ does not correspond to the natural notion of automaton equivalence (as in the Myhill-Nerode theorem [9]) that two automata should produce the same output for a given input string. In general, the compressed automaton represents a weakening of the original finite state machine, in the sense that it will detect monotonically fewer anomalies. For example, if we run the resulting FSM on the test string ABCCABBCCABABC (in which a B is inserted between the two final As of the training string), we get two anomalies (marked with asterisks below) after the last CAB, since A and B are not accepting states. By contrast, the original automaton found three anomalies.

```
C C A B A B C
C C
  C A
  C A B
      A*
        B*
        B C
```

The empirical question is whether the compressed automaton is still able to detect attacks. We will address this question in Section 4.

3.3 Automatic construction without N

Given a value for N, the construction so far enables us to define a set of "N-grams" of different lengths and a compressed automaton that implements the sliding window algorithm. It is, however, not sufficient for achieving our goal, which is to automatically find strings of the "right" length.

Now suppose that we pick a large value of N—not huge, but large enough that we are comfortably assured that it exceeds whatever "good" value of N we would like to end up with. The result will be a self database containing more strings than we need, some of them quite long. Suppose further that two subtrees are similar, in the sense of Figure 4, except that way down in the subtrees there is a difference between them. If the difference is far enough down, then we might wish to treat the two subtrees as if they were equivalent. If we have chosen a large enough value for N, then we do not mind small discrepancies lower down in the subtrees. Many longer strings are simply artifacts anyway—they may well be concatenations of shorter strings that already capture the essence of the training data.

In a suffix automaton based on a suffix tree, let Σ_s^i be the sequences of length i that can occur without causing a transition through a suffix state. They correspond to tree paths in the suffix tree. Σ_s^1 is Σ_s. In our example, Σ_{CA}^1 is {A,B}, Σ_{CA}^2 is {AB, BB}, Σ_{CA}^3 is {ABC, BBC}, and so on. For this weaker notion of equivalence, we require only that Σ_s^i be the same for the two subtrees up to Σ_s^k. Intuitively, the subtrees match down to a depth of k, but they may diverge below that. We call the states corresponding to the roots of the two subtrees *k-similar*. For example, in Figure 2, states CA and A are not similar, because they are not 2-similar (node AB has two out-edges, while node CAB has only one). However, they are 1-similar.

Definition. Any state is *k-similar* to itself. If $s_1 = \varphi(s_2)$ or $s_2 = \varphi(s_1)$, then s_1 and s_2 are k-similar, $s_1 \sim_k s_2$, iff

(a) $k = 0$, or

(b) $\Sigma_{s_1}^i = \Sigma_{s_2}^i$, and

$$\forall x \in \Sigma_{s_1},\ \delta_{s_1}(x) \sim_{k-1} \delta_{s_2}(x).$$

The transitive closure of k-similarity is also an equivalence relation on S and the k-similar states form a suffix automaton in

the same way as the similar states. We call *k* the *approximation parameter*.

Using k-similarity, we can compress the automaton of Figure 2. Figure 6 shows the graph that results from the example suffix tree by compression using an approximation parameter of 1. It differs from the graph of Figure 5 in that there are no nodes corresponding to CA or CAB. All nodes are accepting, except for the root node and [C].

If we run the resulting automaton on the test string ABCCABBCCABABC (in which a B is inserted between the two final As of the training string), we get no anomalies, as shown in the sequence of states below. The suffix link from [A] goes to the root (empty string) node, and thence to state [B], but in this automaton [A] and [B] are accepting.

```
C C A B A B C
C C
    A
      B
        A
          B
          B C
```

In general, the weakened automaton that comes from using k-similarity will find fewer anomalies than either the original automaton or the automaton based on similarity, for the same value of N. However, recall that our plan is to pick a large value of N (rather than N = 4, as in this example). The hope is that even if the resulting automaton cannot detect all anomalies for the large value of N, it will still be able to detect attacks that could be detected for some smaller values of N.

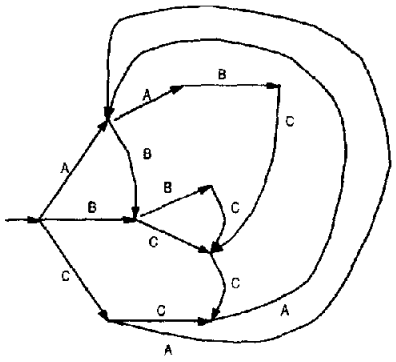

Figure 6. Result of compression using k=1

In the next section, we address the empirical question of how well the resulting detector is able to detect actual attacks.

4. Results

In this section, we compare the effectiveness of this algorithm with the sliding window algorithm and briefly examine its effect on the size of the self database.

Characterizing the program behavior with multiple-length strings is essentially a weakening of the characterization as a set of N-grams (but with a larger N). Because our algorithm is weaker than Forrest's sliding window algorithm and we assume that training continues until the self database converges, our algorithm will not introduce any false positives. That is, it will not find "anomalies" that Forrest's N-gram method does not find. However, we need to show that it is effective in finding anomalies—i.e., does not introduce false negatives.

The algorithm presented here has been programmed and applied to three sets of training data:

- `lpr` training data and exploit from the University of New Mexico [2]
- `inetd` training data and exploit from the University of New Mexico [3]
- data from the PersonnelTracker application collected by the CORBA Immune System [10].

4.1 Synthetic `lpr` data from the University of New Mexico

Figure 7 shows a graph of the *locality frame measure* [8] for an `lpr` exploit, as measured against UNM's "synthetic normal" self database for `lpr`. The locality frame measure counts the number of anomalies in the last L N-grams and is a good indicator of anomaly clusters, which characterize attacks. Following Forrest, we have used L = 20 in this paper. For example, a locality frame measure of 10 means that half of the previous 20 N-grams were anomalous. The spike at the end of the graph corresponds to an `lprcp` attack (the two minor bumps are due to symbols that do not appear in the training data and are not important).

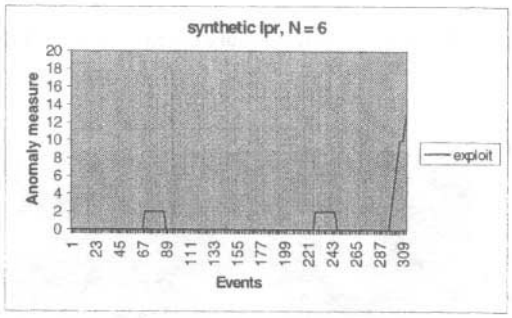

Figure 7. N-gram anomaly measure for `lpr` exploit (N = 6)

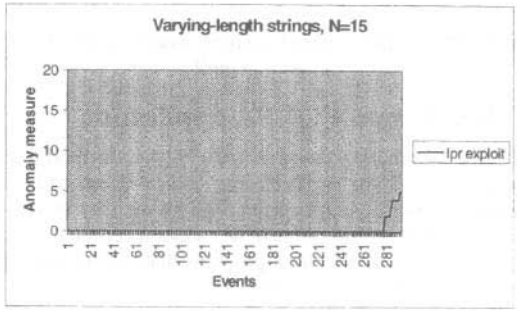

Figure 8. Multiple-length strings (max length = 15) for `lpr` exploit

Figure 8 shows the same exploit when measured against a multiple-length strings self database with a maximum string length of 15. In this and subsequent examples, we use an

approximation parameter of 1. This weakens the detector the most and results in a self database with the shortest strings—in short, it is more apt to result in false negatives than other values of d. Note that the anomaly value reaches a maximum of 13 (out of 20) in Figure 7 but only 5 in Figure 8. Note also that the minor bumps do not appear in Figure 8. In both cases, the data are sufficient to distinguish an attack from a non-attack.

4.2 `inetd` data from the University of New Mexico

Figure 9 and Figure 10 show analogous anomaly measure data for a denial of service attack against `inetd`. The self database was generated from training data collected at UNM. In this example, the maximum anomaly value for the N-gram method is 20, vs. 10 for multiple-length strings. Again, the attack is clearly visible in both.

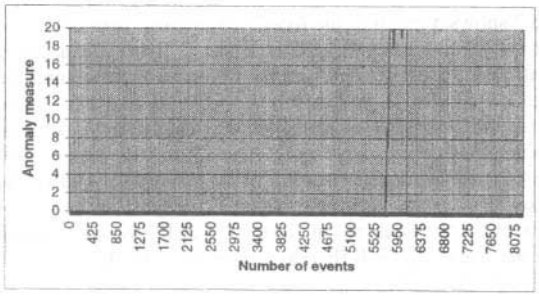

Figure 9. N-gram anomaly measure for `inetd` exploit (N = 6)

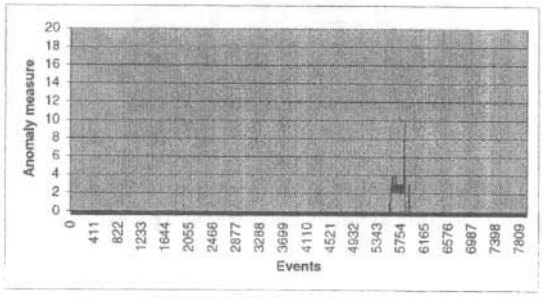

Figure 10. Multiple-length strings (max length = 15) for `inetd` exploit

4.3 Attack on a CORBA application

The CORBA Immune System uses the N-gram method to catch rogue clients. Some applications require client and server programs to work closely together; the server may rely on the client to follow a certain protocol or maintain certain invariants. A rogue client is a malicious program that masquerades as a legitimate client of such a server.

Figure 11 shows the anomaly measure graph of a rogue client, using training data and attack data collected by the CORBA Immune System for the PersonnelTracker application. For this application, N = 3. Note that Figure 11 looks very different from the graphs of the Unix processes. In the `lpr` and `inetd` examples, most of the execution is normal, with an attack occurring after a period of normal execution. In the rogue client type of attack, the entire execution is abnormal, since none of the normal client code is being executed.

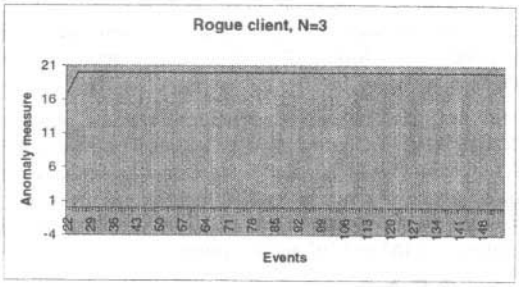

Figure 11. Rogue client, measured using N-gram method (N = 3)

Figure 12 shows the results for the multiple-length string method (maximum length = 6), using the same training and attack data. As in the Unix examples, the difference between an attack and a random anomaly is clear, although all of the anomaly values are smaller.

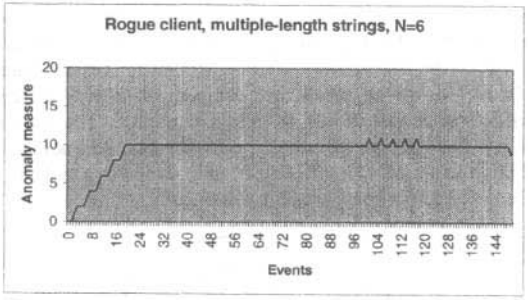

Figure 12. Rogue client, measured using multiple-length strings (N = 6)

An example of a normal trace with a slight anomaly is shown in Figure 13 (multiple-length strings, max length = 6).

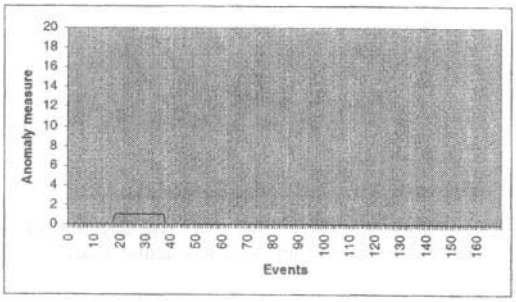

Figure 13. Insignificant anomaly in normal client

4.4 Self database size

For the examples we have tested, multiple-length strings allow the self database to be much smaller than the corresponding N-gram database. For example, using Forrest's `lpr` data, there are 177 strings and 1062 symbols in the self database, while with our method, there are 131 strings and 318 symbols. For

inetd, our method reduces Forrest's 126 strings and 756 symbols to 92 strings and 281 symbols.

Such comparisons are not entirely fair, since one self database is a set and the other a finite state machine. That is, there is some hidden information in the state machine—namely, which symbol(s) can appear next in normal traces—that supplements the information in the strings. Nevertheless, the primary reason for the reduction is the intrinsic redundancy in N-grams, which our method wrings out. The number of transitions in the FSMs is reduced along with the number of strings and symbols in the self database. For example, for `lpr`, the two-finger algorithm requires 774 transitions, as opposed to 307 for multiple-length strings.

In short, our results so far suggest that extra processing at the time of constructing the self database can result in a significantly more efficient detector at run time.

4.5 Self database size as a function of N

For our method to be practical, we would like the size of the FSM to converge as N grows, indicating that there is a "natural" set of covering strings. It also means that in selecting a maximum value for N, there is no penalty for choosing a slightly larger value than necessary. Figure 14 shows that this is indeed the case. As N increases, the number of N-grams (first column) grows, while the number of states in the FSM (second column) levels off after N = 6. The number of multiple-length strings in the database (the number of accepting states in the FSM, shown in the third column) also levels off after N = 6.

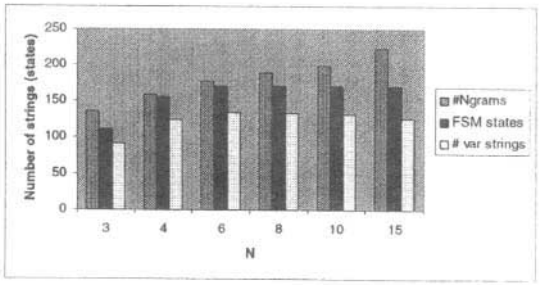

Figure 14. Growth of synthetic `lpr` database as function of N

Figure 15 shows the analogous chart for the growth of the self database in the rogue client example.

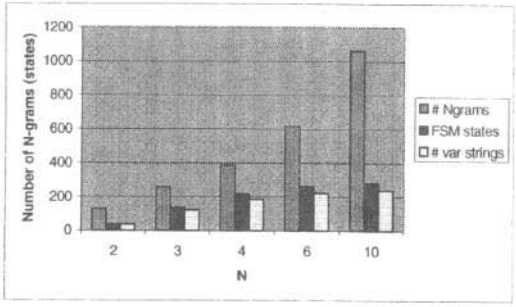

Figure 15. Growth of PersonnelTracker database as function of N

5. Related Work

Hervé Debar has used traditional string processing techniques to find longest repeating patterns in training data [4]. Consider, for example, the string "ABCABCABCABC." Debar's method would characterize that string using the repeating substring "ABC." By contrast, we look for very short strings that are good predictors of the next character. The result is a sliding window that always advances one character to the right, but whose width may vary at each step. As noted above, we would characterize this string by the substrings "A," "B," and "C."

Because of his focus on repeating patterns, Debar's detection method is based on matching *adjacent* substrings. His algorithm, then, hops from one substring instance to the next, rather than sliding from one symbol to the next. At each hop, the test string is tested against the various patterns in the self database.

Like ours, Debar's construction of the self database uses a suffix tree, but instead of pruning to a maximum depth of N, he prunes branches based on relative frequency in the training data. Thus, strings that occur relatively few times are assumed to be rare in practice and are removed from the suffix tree. His method relies on a rareness parameter, as ours does on the approximation parameter.

Debar's input data for both training and detection consists of audit events, rather than the kernel calls used by Forrest. The audit events are further preprocessed to remove consecutive occurrences of the same call—hence any two adjacent symbols in the processed data are guaranteed to be different. Given this input data and his adjacent string detection method, he has found that his variable-length strings are just as good at detection as his fixed-length strings. Because of the differences in input data, preprocessing, and detection method (hop vs. skip), it is difficult to compare Debar's results (for either fixed-length or variable-length strings) with ours or with those obtained by the University of New Mexico group.

6. Future Research

The results that we have presented here are very preliminary. In order to demonstrate its general applicability, our method should be applied to many other examples, both to further examples of Unix processes and to examples from other domains to which the N-gram method has been applied.

Previous work in Computational Immunology has been criticized on the grounds that it relies on a "magic number"—that there are no compelling reasons for using six as the value of N. Our results suggest that six is indeed a natural value for N *in sequences of Unix kernel calls*, since the number of multiple-length strings resulting from our method stops growing after N=6 (see Figure 14). It would be interesting to study more examples to find out whether they all support the same value, or whether the value varies significantly depending on the program.

Finally, in the examples we have studied, an approximation parameter of 1 (k = 1) was adequate for detection. These examples were characterized by excellent coverage (that is, the training data covered all program behavior). In larger applications, complete coverage may be difficult or too expensive to obtain. Larger approximation parameters give a closer approximation to the original sliding window algorithm.

It may be possible to compensate for poorer coverage by using larger approximation parameters. To date, we have little experience with values greater than one.

7. Conclusions

We have presented a way of constructing a self database for a program in the form of a finite state machine. The finite state machine implicitly defines a set of multiple-length strings that cover the empirical training data. The primary advantage of this method over N-grams is pragmatic: it is not necessary to find a suitable value for N. The method also provides an implementation technique that generates efficient detectors. The resulting self database is less sensitive to anomalies than the N-gram database but it appears to have sufficient sensitivity in practice to detect attacks.

8. Acknowledgements

The two-finger finite-state-machine implementation of Forrest's sliding-window algorithm was developed jointly with Matthew Stillerman and Francis Fung of ORA. David Rosenthal, Ira Moskowitz, and anonymous reviewers provided helpful criticisms of earlier versions of this paper.

9. References

[1] Arbib, M.A., *Theories of Abstract Automata.* 1969, Englewood Cliffs, NJ: Prentice-Hall.

[2] Computer Science Department University of New Mexico, *Synthetic UNM lpr data.* 1995: http://www.cs.unm.edu/~immsec/data/synth-lpr.html.

[3] Computer Science Department University of New Mexico, *UNM live inetd data.* 1996: http://www.cs.unm.edu/~immsec/data/live-inetd.html.

[4] Debar, H., Dacier, M., Nassehi, M., and Wespi, A., "Fixed vs. variable-length patterns for detecting suspicious process behavior," in *ESORICS 98, 5th European Symposium on Research in Computer Security*, 1998, Louvain-la-Neuve, Belgium: Springer Verlag.

[5] Forrest, S., Hofmeyr, S., and Somayaji, A., "Computer immunology," *Communications of the ACM*, 1997, **40**(10), p. 88-96.

[6] Forrest, S., Hofmeyr, S.A., and Somajayi, A., "A Sense of Self for UNIX Processes," in *1996 IEEE Symposium on Computer Security and Privacy*, 1996: IEEE Press.

[7] Gusfield, D., *Algorithms on Strings, Trees, and Sequences.* 1997: Cambridge University Press.

[8] Hofmeyr, S., Forrest, S., and Somayaji, A., "Intrusion detection using sequences of system calls," *Journal of Computer Security*, 1998, **6**, p. 151-180.

[9] Nerode, A., "Linear automaton transformations," *Proceedings of the American Mathematics Society*, 1958, **9**, p. 541-544.

[10] Stillerman, M., Marceau, C., and Stillman, M., "Intrusion Detection for Distributed Applications," *Communications of the ACM*, 1999, **42**(7), pp. 62-69.