A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors

R. Sekar M. Bendre D. Dhurjati
State University of New York
Stony Brook, NY 11794
{sekar,mbendre,dinakar}@cs.sunysby.edu

P. Bollineni
Iowa State Univeristy
Ames, IA 50014
pradeep@cs.iastate.edu

Abstract

Forrest et al introduced a new intrusion detection approach that identifies anomalous sequences of system calls executed by programs. Since their work, anomaly detection on system call sequences has become perhaps the most successful approach for detecting novel intrusions. A natural way for learning sequences is to use a finite-state automaton (FSA). However, previous research seemed to indicate that FSA-learning is computationally expensive, that it cannot be completely automated, or that the space usage of the FSA may be excessive. We present a new approach in this paper that overcomes these difficulties. Our approach builds a compact FSA in a fully automatic and efficient manner, without *requiring access to source code for programs. The space requirements for the FSA is low — of the order of a few kilobytes for typical programs. The FSA uses only a constant time per system call during the learning as well as detection period. This factor leads to low overheads for intrusion detection. Unlike many of the previous techniques, our FSA-technique can capture both short term and long term temporal relationships among system calls, and thus perform more accurate detection. For instance, the FSA can capture common program structures such as branches, joins, loops etc. This enables our approach to generalize and predict future behaviors from past behaviors. For instance, if a program executed a loop once in an execution, the FSA approach can generalize and predict that the same loop may be executed zero or more times in subsequent executions. As a result, the training periods needed for our FSA based approach are shorter. Moreover, false positives are reduced* without *increasing the likelihood of missing attacks. This paper describes our FSA based technique and presents a comprehensive experimental evaluation of the technique.*

1. Introduction

Forrest et al [5] demonstrated that effective intrusion detection techniques can be developed by learning normal program behaviors, and detecting deviations from this norm. In contrast with users, programs tend to have more narrowly-defined behaviors. This enables more accurate learning of normal behaviors, and thus improves the accuracy of intrusion detection.

Forrest et al's [5] approach characterizes normal program behaviors in terms of sequences of system calls made by them. Anomalous program behavior produces system call sequences that have not been observed under normal operation. In order to make the learning algorithm computationally tractable, they break a system call sequence into substrings of a fixed length N. These strings, called N-grams, are learnt by storing them in a table. In practice, N must be small ([5] suggests a value of 6) since the number of N-grams grows exponentially with N. Figure 1 illustrates the N-grams associated with a simple program, where a value of $N = 3$ has been used for illustrative purposes.

A drawback of using small values of N is that the learning algorithm becomes ineffective in capturing correlations among system calls that occur over longer spans. For instance, the program in Figure 1 will never produce the sequence $S_0S_3S_4S_2$. However, the trigrams in this sequence ($S_0S_3S_4$ and $S_3S_4S_2$) are produced by the program, and hence the N-gram learning algorithm would treat this sequence as normal. The second difficulty with the N-gram algorithm is that it can recognize only the set of N-grams encountered during training; similar behaviors that produce small variations in the N-grams will be considered anomalous. [8] reports that this lack of generalization in the N-gram learning algorithm leads to a relatively high degree of false alarms.

An alternative approach for learning strings is to use finite-state automata (FSA). Unlike the N-gram algorithm which limits both the length and number of sequences, an FSA can capture *an infinite number of sequences of arbitrary length* using finite storage. Its states can remember short *and* long-range correlations. Moreover, FSA can capture structures such as loops and branches in programs — by traversing these structures in different ways, it is possible to produce new behaviors that are similar (but not identical) to behaviors encountered in training data. In spite of these advantages, experience with finite-state-based learning has been mostly negative:

```
1.  S0;
2.  while (..) {
3.      S1;
4.      if (...) S2;
5.      else S3;
6.      if (S4) ... ;
7.      else S2;
8.      S5;
9.  }
10. S3;
11. S4;
```

$S_0S_1S_2$	$S_1S_2S_4$	$S_2S_4S_5$	$S_3S_4S_5$	$S_4S_5S_1$	$S_2S_5S_1$	$S_5S_1S_2$
$S_0S_1S_3$	$S_1S_3S_4$	$S_2S_4S_2$	$S_3S_4S_2$	$S_4S_5S_3$	$S_2S_5S_3$	$S_5S_1S_3$
$S_0S_3S_4$				$S_4S_2S_5$		$S_5S_3S_4$

Figure 1. An example program and associated trigrams. S0,...,S5 **denote system calls.**

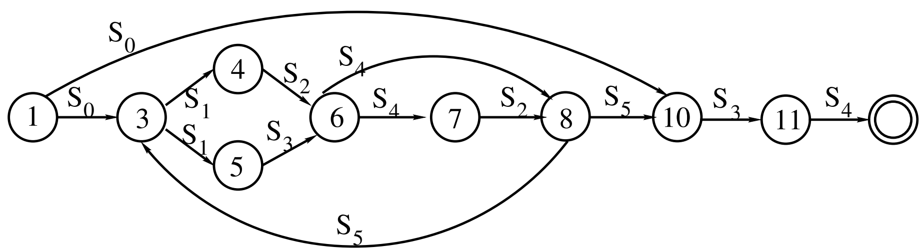

Figure 2. Automaton learnt by our algorithm for Example 1

- Several researchers [25, 14] have shown that the problem of learning compact FSA is hard. For instance, [14] show that learning approximately optimal FSA is as hard as integer factorization.

- [16] describe a methodology for learning system calls using finite-state automata. However, no algorithm is provided for constructing FSAs from system call traces. Instead, they rely on human insight and intuition to construct FSA states and edges from sequences.

- [30] studied several learning algorithms, including those based on the Hidden Markov Models (HMM) [26] that are similar to FSA. In their experiments, HMMs incurred large overheads for learning, while improving detection accuracy over the N-gram algorithm only slightly.

Against this backdrop of negative results regarding FSA-based learning, we present a new, positive result: *Compact FSAs characterizing process behaviors can be learnt fully automatically and efficiently.* Whereas [30] concluded that the N-gram algorithm provides the best overall performance among many different algorithms, our results show that the FSA-algorithm further improves detection and training performance significantly. Below we provide an overview of the FSA-based learning algorithm and summarize its benefits.

1.1. Overview of FSA Algorithm and its Advantages

The central difficulty in learning an FSA from strings is that the strings do not provide any direct information about internal states of the automaton. For instance, if we observed an execution of the program in Figure 1 and witnessed a sequence of system calls $S_0S_1S_2S_4S_2...$, we would not know whether to treat the two occurrences of S_2 to be from the same automaton state or not. It is this key problem that leads to the difficulties in efficient learning of automata from string examples.

The key insight behind our technique is that we can indeed obtain state-related information if we knew the *program state* at the point of system call; and that the very same operating system mechanisms that can be used to trace system calls can also be used to obtain the program state information. When the above system call sequence is augmented with point-of-system-call information, we obtain:

$$\frac{S_0}{1}\frac{S_1}{3}\frac{S_2}{4}\frac{S_4}{6}\frac{S_2}{7}...$$

Based on the program state information, the FSA-algorithm will learn the automaton shown in Figure 2 from the above program. The example provides the basis to illustrate the advantages of the FSA-algorithm.

- *Faster learning.* The following two execution sequences suffice for learning the complete automaton shown in Figure 2. In contrast, they contribute only 11 of the 17 trigrams (65%) learnt by N-gram algorithm.

  - $\frac{S_0}{1}\,\frac{S_3}{10}\,\frac{S_4}{11}$
  - $\frac{S_0}{1}\,\frac{S_1}{3}\,\frac{S_2}{4}\,\frac{S_4}{6}\,\frac{S_5}{8}\,\frac{S_1}{3}\,\frac{S_3}{5}\,\frac{S_4}{6}\,\frac{S_2}{7}\,\frac{S_5}{8}\,\frac{S_3}{10}\,\frac{S_4}{11}$

  In our experiments, FSA learning converged *an order of magnitude* faster than the N-gram learning.

- *Better detection.* Using program counter information, it is possible to detect some classes of attacks that elude algorithms that do not utilize such information. (See Section 4.5 for further discussions.) Even without the program counter information, the state-sensitive nature of the FSA-algorithm will enable detection of attacks missed by the N-gram algorithm. For instance, the trigrams in the system call sequence $S_0S_3S_4S_2$ all occur during normal execution of the above program, and hence the N-gram algorithm cannot detect this sequence as anomalous. However, the FSA-algorithm will detect that the program does not produce this sequence.

- *Reduction in False Positives.* Reduction of false positives depends upon the ability of a technique to generalize past behavior to predict future behavior. In particular, on seeing the second of the above execution sequences, the FSA-algorithm is able to learn the branching structure of the program, and is able to predict that these branches may be combined in other ways, leading to an infinite set of strings such as:

  - $\frac{S_0}{1}\,\frac{S_1}{3}\,\frac{S_3}{5}\,\frac{S_4}{6}\,\frac{S_5}{8}\,\frac{S_3}{10}\,\frac{S_4}{11}$
  - $\frac{S_0}{1}\,\frac{S_1}{3}\,\frac{S_2}{4}\,\frac{S_4}{6}\,\frac{S_2}{7}\,\frac{S_5}{8}\,\frac{S_3}{10}\,\frac{S_4}{11}$

- *Compact representation.* Finite-state automata provide a very compact way to represent the large (typically infinite) set of execution traces that can be produced by a program. For instance, the trigram representation needs to represent 51 system calls in the model. The corresponding measure in the automaton is the number of edges in it (with each edge being labelled with a system call), and this number is only 13. Our experiments show that a factor of 3 to 4 reduction in space utilization over the 6-gram algorithm. (We note that in absolute terms, space requirements are modest for both the N-gram and the FSA-algorithms.)

- *Fast detection.* Intrusion detection using the FSA model requires matching system call sequences using the FSA. It is clear that matching using the FSA takes constant time per system call, and this time is fairly small (less than a hundred instructions). In contrast, each system call execution typically involves several hundreds of instructions, thus the overhead of matching using the automaton is small.

1.2. Related Work

Intrusion detection techniques can be classified into two classes: *misuse detection* and *anomaly detection*. Misuse detection techniques [29, 23, 17] model known attacks using patterns (also known as signatures), and detect them via pattern-matching. Their benefit is a high degree of accuracy, and their main drawback is the inability to identify novel attacks. Anomaly detection techniques [1, 5, 20, 24, 4, 8] address this problem by flagging any abnormalities in user or system behavior as a potential attack. One of the main research problems in anomaly detection is that of learning normal user or system behaviors. We focus our discussion below on anomaly detection techniques most closely related to our approach.

Approaches Based on Learning Program Behaviors. The use of system call sequences to model program behaviors was first suggested by Forrest et al [5]. [16] proposes to increase the accuracy of the N-gram learning algorithm by using an FSA representation. However, no algorithm is provided for FSA construction; instead, a manual procedure is employed. [18] describes an algorithm for constructing finite-state automata from strings, but their algorithm treats only strings of a finite length. Thus, their approach learns tree-structured automata. The problem of learning tree automata is computationally much simpler than a general FSA that contains cycles.

[30] studies four different algorithms for learning program behaviors. Of particular interest was a data-mining based algorithm suggested in [20]; and the Hidden Markov Model (HMM), which is a finite state model widely used in speech recognition. They concluded that HMMs provide slightly increased accuracy, but the length of training required made them unattractive for intrusion detection. Their overall conclusion was the the N-gram algorithm provides the best combination of low training periods, high detection rates and low false positives. As compared to these algorithms, the FSA learning algorithm possesses the following advantages:

- It does not limit the length or number of system call sequences: entire sequence produced by each run of a program is learnt by the FSA. This factor will likely contribute to more accurate intrusion detection.

- It captures the branching and looping structures of the program, thus enabling us to recognize typical variations in behaviors of programs. This factor will likely reduce false positives.
- It is capable of learning program behaviors while "leaving out" behaviors captured by library functions. This can lead to smaller storage requirements. It can also contribute to shorter training periods since we do not waste time in learning the behavior of libraries.

Static Construction of FSA. We note that the FSA learnt by our approach captures program structures that are similar to those captured by control-flow graphs used in compilers. Thus it is possible to develop compile-time analysis techniques to learn the FSA *statically,* without any runtime training. A disadvantage is that interprocedural analysis, especially in the presence of libraries that are dynamically linked (and hence unavailable at compile time) poses nontrivial problems. An alternative is to develop link-time analysis of object files and libraries to construct the FSA. We are currently studying this approach. Even if this approach were to be successful, runtime construction, as proposed in this paper, would still have additional information to offer. In particular, a learning algorithm that constructs the FSA at runtime can incorporate information about frequency of execution. This information is unavailable in a compile-time or link-time approach.

2. Learning Finite-State Automata

Our learning algorithm is based on tracing the system calls made by a process under normal execution. As each system call is made, we obtain the system call name as well as the program point from which the system call was made (given by the value of the program counter (PC) at the point of system call). Each distinct value of the program counter corresponds to a different state of the FSA. The system calls correspond to transitions in the FSA. To construct the transitions, we use both the current pair of $\frac{SysCall}{PC}$, and the previous pair, $\frac{PrevSysCall}{PrevPC}$. The invocation of the current system call $SysCall$ results in the addition of a transition from the state $PrevPC$ to PC that is labelled with $PrevSysCall$. The construction process continues through many different runs of the program, with each run possibly adding more states and/or transitions. Figure 3 illustrates this process.

The simple algorithm outlined above can deal with statically linked programs, but does not always work for dynamically linked programs. The key difficulty is that the value of program counter cannot be relied upon, as the same functions may get loaded at different locations in a dynamically linked program. One may try to use relative values of program counters instead of absolute values, but this does not work either: the relative locations of functions across two different libraries can vary from one run to another.

The second difficulty is that most programs make heavy use of library functions, which in turn make several system calls. For instance, consider a simple program:

```
main() {
  int ch;
  while ((ch = fgetc(stdin)) >= 0)
    fputc(ch, stdout);
}
```

It would be better to capture the behavior of this program as consisting of read and write system calls made from the main program. However, if we used the program counter value at the time of actual system call, no information about the structure of the main program will be captured. Instead, we would be capturing the structure of the library functions — in fact, since every "system call" invocation is actually made from within a library function within `libc`, the automaton will capture no useful information about the structure of the main program. As a result, the automaton learnt will remain very similar across different programs, since library code used by most programs are identical. In order to capture the behavior of the program, it is necessary to record the location from where the library function was called, rather than recording the location within the library code from where a system call was made. We describe our approach for doing this below, after a brief discussion of the system call interception mechanisms we use.

2.1. System Call Tracing

Several approaches have been proposed for system call tracing over the past several years. Some of these techniques involve modifications to the operating system kernel, as in [7, 6, 19]. The primary benefit of a kernel-based approach is speed, while its disadvantage is the need to modify the kernel. Other approaches such as [13] make use of the process tracing capability provided by most versions of UNIX in order to perform system call interception at the user level. We used the second approach in this work.

Most versions of UNIX provide a mechanism by which one process can trace the system calls made by another process. Programs such as `strace`, `truss` and `par` utilize the low level OS mechanisms and provide a command line interface for recording system calls. Previous research, such as [5], utilized such programs to record system calls in a log file, and then used an offline learning algorithm. In our approach, we directly make use of the OS mechanisms. The key benefits are that we are able to use additional information (e.g., the contents of the registers and the stack of the traced process) that is available at the level of the OS-provided mechanisms, but not made available by the above-mentioned applications.

- $\frac{sc1}{pc1}\frac{sc2}{pc2}\frac{sc3}{pc3}$
- $\frac{sc4}{pc1}\frac{sc3}{pc3}\frac{sc4}{pc1}\frac{sc3}{pc3}$

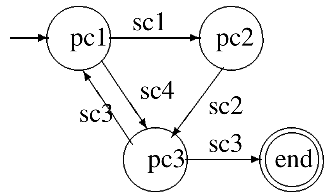

Figure 3. Two traces produced by a program and the generated automaton

2.2. Keeping Track of Different Sections of Code

The general problem is to trace back each system call to the innermost function call that was made from certain regions of memory. Note that most libraries are linked and loaded dynamically, and that the non-library components are statically linked. We therefore trace back all system calls to statically linked code sections.

The first step in tracing back is to identify code sections that are statically linked. Our approach for doing this relies on (a) the structure of the ELF (Executable and Linking Format) format used in Linux and most other UNIX systems, and (b) tracing system calls used to load the dynamically linked libraries. The range of addresses of the statically linked code segment is obtained from the header information in the executable file. For the addresses of dynamically linked regions, we note that in Linux, the dynamically linked code is loaded using the *mmap* system calls. From the return value of this system call, and the size argument provided to this system call, we can obtain the addresses corresponding to the dynamically linked libraries.

2.3. Stack Traversal

Procedure calls are implemented using a process stack. The stack is partitioned into many *activation frames,* each of which correspond to an invocation of a procedure. The innermost active procedure invocation corresponds to the top-most frame on the stack. An activation record stores information such as the return address, procedure parameters and local variables of the procedure. Both the caller and the called procedures need to access the return address and parameters. Hence the structure of the activation records as well as the location of these fields within the activation record are standardized, even across different programming languages.

Based on the above structure of the stack, tracing back of the system call can proceed as follows. We examine the value of the program counter (which is saved by the processor when the trap instruction to switch to the kernel mode was executed) and see if it is from the statically linked portion of the executable. If so, we are done. Otherwise, we examine the topmost frame on the stack, and extract the return address information. If this address corresponds to a statically linked region of the program, we are done. Otherwise, we move to the next stack frame (corresponding to the next outer procedure invocation) and repeat the same process.

We observe that this approach will not work satisfactorily if the statically linked portion of the code itself contains library functions, or wrapper functions that have been introduced for portability. In those cases, the FSA will learn the location within the library from where a system call is made.

2.4. Dealing with `fork/exec`

The `fork` and `exec` system calls require special attention, since they create copies of a running process or change it altogether. A corresponding change has to be made to the FSA being learnt for the program.

The fork system call causes the process to create a copy of itself. We use the same FSA to capture the behavior of the child as well as the parent. Unless the fork system call is followed by `execve`, the child process usually performs the same tasks as that of the parent (e.g., servicing more requests in a http server) and so this can be justified. After the fork, subsequent system calls made by either the parent or the child is added as a transitions to the same FSA. This requires us to keep track of all the current states corresponding to the parent and all of the children processes. When one of these processes makes a system call, an edge is added from the current state of this process. At intrusion detection time, we follow a similar procedure.

When an `execve` system call is made, we need to decide whether the system calls of the new program (to be executed) are to be learnt using the same FSA, or to use a different FSA. In the former case, an FSA that is customized for this particular execution of the new program is created. This would enable us to capture, for instance, that when a program A executes another program B, it uses B's functionality in a restricted way. For instance, a program may spawn a shell, which may in turn be used to execute a specific script; but the full functionality of shell is not accessed. In the latter case, we retrieve the FSA that has been learnt so far for the program `execve`'d, and start augmenting this FSA to incorporate the sequence of system calls observed in the current execution. Currently, we use this second option as the default.

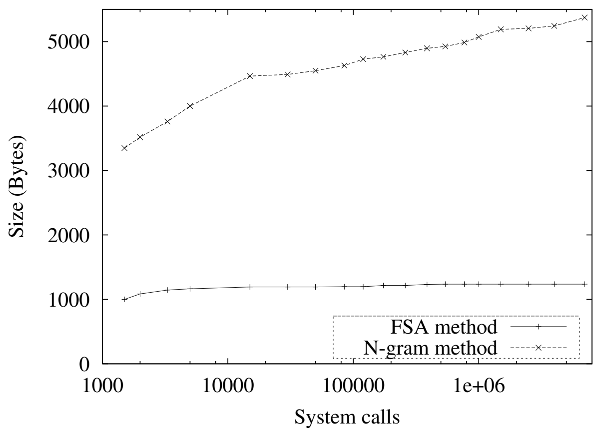

Figure 4. Convergence on NFS Server.

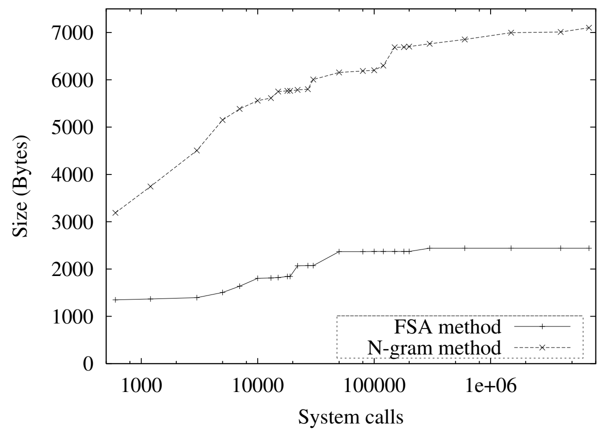

Figure 5. Convergence on FTP Server

3. Runtime Monitoring for Intrusion Detection

Matching runtime behavior to the automaton proceeds as follows. At any point during runtime, the state of the match would be captured by a *current state* of the FSA. For each system call intercepted, we proceed as follows:

- Obtain the corresponding location (within the statically linked section of the program) from where the call was made. If an error occurs while doing this, it would be because the stack has been corrupted, possibly due to a buffer-overflow attack.
- Check if there exists a transition from the current state to the new state that is labelled with the system call name that was intercepted. If not, there is again an anomaly. Anomalies of this kind may arise either due to attacks, or because of unusual behavior of the program that had not been observed during learning.
- Update the state of the automaton to correspond to the new state. If the new state is not in the automaton, transition to a "sink" state in the FSA[1].

To ensure that isolated mismatches do not immediately result in an intrusion being flagged, a leaky bucket algorithm is typically used (as in [5, 9]) to aggregate anomalies over time. Each time an anomaly is detected, an anomaly count is incremented. When the anomaly count exceeds a threshold, an intrusion is flagged. The anomaly count is decremented periodically, which has the effect of ignoring isolated anomalies.

[1]Note that the FSA is not "stuck" in the sink state: as soon as the program execution returns to a location that had been observed during learning, the automaton would transition to that state from the sink state. Thus, the use of program counter information enables the automaton to "resynchronize" with the program even if synchrony is lost momentarily due to execution of new code.

Several different kinds of anomalies are recognized by the method described above. Our method associates different *weights* with different kinds of anomalies. Instead of incrementing the anomaly count by one, we increment it by the weight associated with the anomaly observed. The weight associated with stack corruption anomaly is set to be high enough that even a single occurrence of the anomaly will be flagged as an intrusion. The weight associated with a missing program state is smaller, such that several successive occurrences of these anomalies must occur before the threshold for flagging an intrusion is reached. The weight associated with a missing transition is higher if the system call corresponding to the transition appears nowhere in the FSA. Otherwise, the anomaly weight is set to be the same as that of a missing state.

4. Experimental Evaluation

To evaluate the FSA-based algorithm, we considered security-critical server programs such as *ftpd, httpd, nfsd* and `telnetd`. Telnetd was later eliminated since its behavior was extremely simple and predictable. Among the other three, ftpd appeared to have the most complex behavior, supporting 70 different operations. nfsd was moderately complex, supporting 17 operations. httpd supports only a small number of commands, but is comparable in complexity to NFS server. Our evaluation addresses the following metrics and compares them with those of the N-gram algorithm: convergence of learning, false positive rate, runtime and space overhead, and attack detection efficacy.

All the results presented in this section were obtained for Linux running on a 266MHz Pentium II processor with 32MB RAM and 3GB EIDE disk. For comparison purposes, we implemented the N-gram algorithm from [5]. This implementation uses a *trie* data structure, which is the most compact data structure for representing large collections of fixed-length strings.

We used the following procedure for conducting these experiments. Most of our experiments were conducted using training scripts that attempt to simulate the requests likely to be handled by each of these servers. Some experiments involving the http server were conducted on a live web server handling requests. While it would have been better to run all of the tests on live servers, such an approach was impractical for us because we did not have access to systems that experienced large enough volumes of traffic to enable us to conduct such experiments. We present our results on live servers in Section 4.4, while the following three sections discuss results obtained using training scripts.

4.1. Convergence

We measured convergence in terms of the space required for storing the automaton (for the FSA-algorithm) or the N-grams (for the N-gram algorithm). These figures were plotted against the number of system calls made by the program being learnt. The graphs use a linear scale on the Y-axis (size of automata or N-gram storage) and a logarithmic scale on the X-axis (number of system calls). In comparing the two algorithms, the actual Y-axis values are not important: what matters for convergence is whether the curves flatten out quickly.

For these experiments, we used training scripts that generated commands to exercise the servers. The training scripts for FTP and NFS were locally developed, while we used the WebStone benchmarking suite to exercise HTTP server. These scripts generate a random sequence of mostly valid commands, interspersed with some invalid commands. These commands involve files of sizes ranging from 500 bytes to 5MB. The distribution of these commands (and file sizes) is set to mimic the distributions observed under normal operation.

The training scripts were used to generate larger and larger sequences of commands in successive runs. The server behavior observed during each run was learnt using the FSA and N-gram algorithms. The initial run included very few commands, typically resulting in about a thousand system calls made by the server. The final run was about 8 million system calls.

4.1.1 Discussion

Rate of convergence is an important factor that governs the amount of training time needed to achieve a given level of false positives. The slower the convergence rate, the longer the training time would need to be.

For all three servers, the FSA algorithm converged around a few hundred thousand system calls, and did not learn any thing new even when the number of system calls

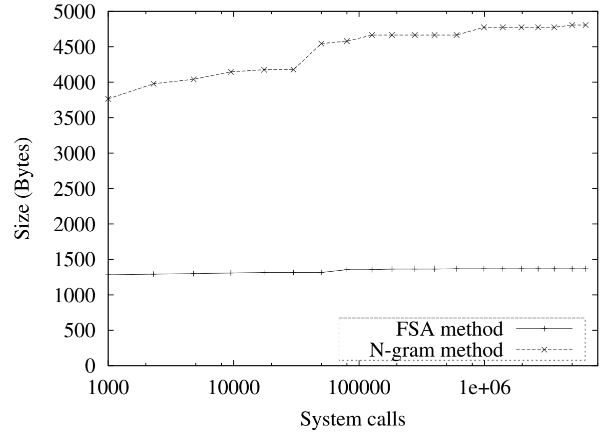

Figure 6. Convergence on HTTP.

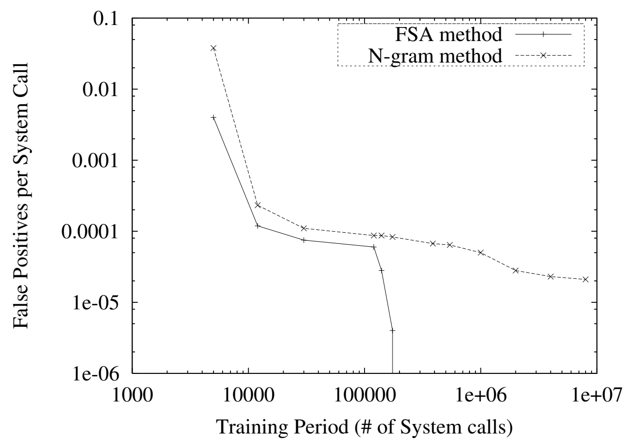

Figure 7. False Positives on NFS Server.

was increased by an order of magnitude beyond this point. The N-gram algorithm converges much more slowly.

Faster convergence of the FSA algorithm is due to two factors. First, the FSA algorithm learns the branching and looping structures in the program. As illustrated with an example in Section 1.1, this factor enables program behaviors to be learnt in fewer runs. The second reason is due to the fact that our algorithm does not preserve the order of system calls made from libraries. For instance, if a library function f is called by the program from a location L, the FSA would contain several edges from L to itself, each labelled with one of the system calls made by f. As a result, variations in the order of system calls made from libraries will not produce changes to the FSA.

4.2. False Positives

To determine false positives, we trained the system with system call traces of different lengths, starting from about

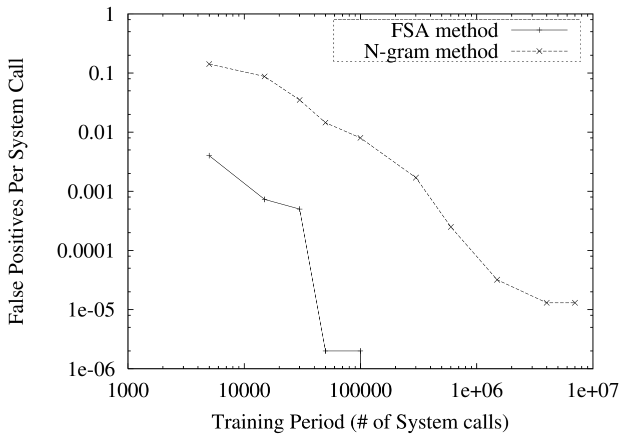

Figure 8. False Positives on FTP Server

5K and ending at about 8M system calls. After training with each trace, the system was run in a detection mode against another system call trace consisting of between 1 and 10M system calls. This trace was produced with the same program as used for training, but with a slightly different distribution of commands (and file sizes). This was done to account for the fact that things can (and typically do) change between the learning and detection times. The exact same system call traces were used to train and analyze the FSA and N-gram algorithms.

For the FSA algorithm, each occurrence of a state or edge that was not present in the FSA was treated as a false positive. For the N-gram algorithm, each occurrence of a new N-gram (which has not been learnt during training) was counted as a false positive. Clearly, more sophisticated thresholding techniques (such as the leaky bucket algorithm) could be used to detect attacks while reducing false positives. However, there is no easy way to choose the parameters, such as the threshold value. Moreover, the optimal parameter values would likely be different for the two algorithms. Rather than spending our efforts in a search for thresholding techniques optimized for each of these algorithms, we decided to use this simpler measure.

Figures 7 and 8 show the number of false positives reported by each algorithm. It shows that the FSA algorithm uniformly produces fewer false positives than the N-gram algorithm. The false positive rate of the FSA algorithm falls below 10^{-6} after a training period corresponding to about 10^5 system calls. The N-gram algorithm continues to produce false positives at a higher rate (in the range of 10^{-4} to 10^{-5}) even after training with over 10^6 system calls.

4.3. Runtime and Space Overheads

Figure 9 shows the runtime storage requirements for representing the behavior learnt by the FSA and N-gram algorithms for the three servers. The figure shows that both algorithms are economical in terms of space usage. FSA-based algorithm improves on the space utilization of the N-gram algorithm by about a factor of four.

Application	N-gram Algorithm	FSA-algorithm
FTP	7.1	2.4
HTTP	4.8	1.4
NFS	5.4	1.2

Figure 9. Space Requirements (in KB) for N-gram and FSA-based algorithms.

To measure runtime overheads, we first split the overhead into two parts: (a) overhead due to execution of learning and/or detection code, and (b) overhead due to system call interception. We measured the two components independently. The overhead due to execution of learning/detection code was between 3% and 4% for all of the three applications. The overhead for the N-gram based algorithm was also about 3%.

The overhead due to system call interception is dependent on the mechanism used for this purpose. Techniques that intercept system calls within the kernel introduce low overheads. User-level mechanisms for interception of system calls, such as the one used by [5] and us, incur significantly higher overheads. This is because of additional task switches required (between the server process and another process that is intercepting its system calls) for each system call. Moreover, every access to server process memory by the monitoring process (between 3 to 8 such accesses are made by the FSA learning algorithm) incurs the overhead of a system call. As a result, the overhead due to system call interception in our implementation is as high as 100% to 250% in terms of CPU time. An `strace`-based implementation such as that used by Forrest et al in their N-gram learning algorithm, introduces overheads in the same range (100% to 250%).

4.4. Results on Live HTTP Server

In this section, we present the results of a comparative experiment involving a live web server. This experiment was performed on http server of the Secure and Reliable Systems Laboratory at SUNY, Stony Brook (`http://seclab.cs.sunysb.edu/`). This site runs an apache web server, and experiences of the order of 3000 hits a day. The web site consists predominantly of passive HTML and image files. A minority of requests involve user authentication, forms and CGI scripts.

One of the difficulties in using a live web server is that the experiment can no longer be conducted in a controlled setting. The requests processed by a live server can vary widely from one day to the next, and thus, we cannot com-

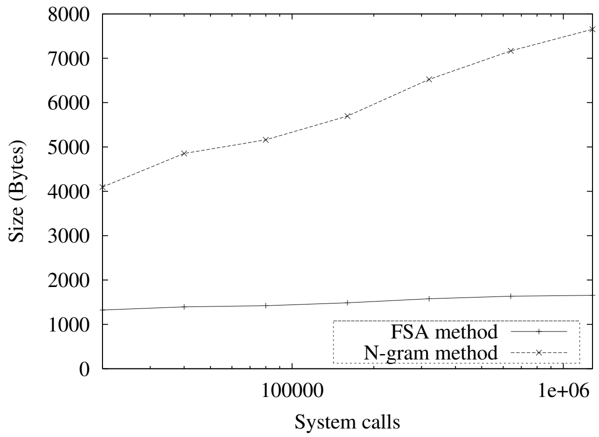

Figure 10. Convergence on Live HTTP Server

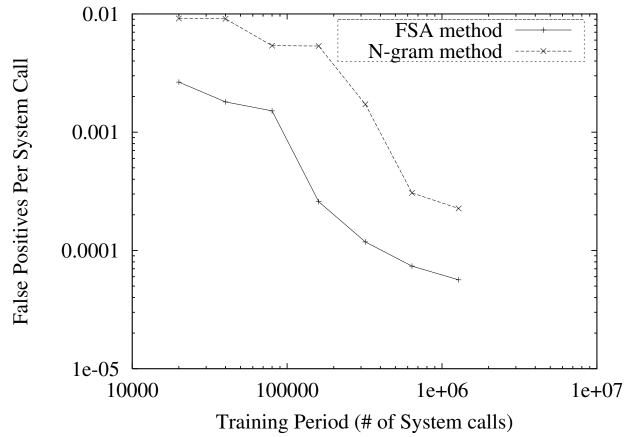

Figure 11. False Positive Rate on Live HTTP Server

pare false alarms observed on one day with that observed on the next day. We therefore decided to run the N-gram and FSA algorithm side-by-side, so that they both make use of the exact same data.

In our experiments, we trained each system for a particular number of system calls, and then ran the system for an extended period of time to compute the false positives rate. The training period was gradually increased, and the false positive rates were plotted as a function of the training period.

One would expect that the false positive rate would fall monotonically with the increases in training period. Observe, however, that with a live web-server, this need not be true. It is possible that the web server received many different kinds of requests on the first day of training, when we used a training sequence of 20,000 system calls. On the second day, we may use a training sequence of 40,000 system calls, but it may turn out that the requests received on the second day were all very similar. As a result, it is possible that more of the server behavior was learnt after the 20,000 system calls seen the first day, as compared to what was learnt after 40,000 system calls the second day. If this were to happen, it will lead to anomalies in the graphs, which would make it very difficult to understand the convergence or false positive rates of these algorithms. To avoid such an anomaly, we used the following approach. The first 20,000 system calls were used to learn a (FSA or N-gram) model. A copy of this model was made, and it was frozen. Subsequent system calls were learnt by the original model until we reached 40,000 system calls. At this point, another copy was made and frozen, while the original model continued to learn subsequent system calls. This process was continued until we processed about 1.5 million system calls. Each frozen version of the model was used to perform false positive analysis on system calls made by the server after the point of freezing.

This approach meant that at any time, a system call made by the web server was processed by seven copies of the N-gram algorithm and by another seven copies of the FSA-algorithm. The entire process was repeated once more, and the results were averaged. (Given the rate of requests received at our web server, this experiment took about three weeks to complete.)

The results of our experiment is shown in Figures 10 and 11. In terms of space requirements, we found that the FSA-algorithm used 1.6KB, while the N-gram algorithm used 7.3KB. Note that these results obtained with the real web server are generally similar to those observed with training scripts. In particular, the difference in the rate of convergence is similar to those observed before. Similarly, the differences observed in false positive rates are similar — the false positive rate of the FSA-algorithm is between 6 to 30 times lower than that of the N-gram algorithm. However, there are some differences as well: the ratio of false positive rates does not increase with the training period, as was observed in the previous experiments. Moreover. the absolute values of false positive rates are much higher. We attribute this to the fact that there is more variation in live traffic than what could be simulated using training scripts. This means that both approaches produce higher false positives. Moreover, if new types of requests that were never experienced before arrive at the web server, it is likely that both approaches would generate the same number of false positives. Thus, the ratio does not increase as with the training script based approach.

4.5. Attack Detection

- *Buffer overflow attacks.* Almost all buffer overflow attacks involve execution of system calls by code running in the stack segment. Our approach will always detect such attacks since it will observe a corrupted stack frame. We have verified this assertion experimentally with a version of FTP server that was modified to introduce new buffer overflow vulnerabilities. Even stealthy attacks that do not execute system calls from the stack could be detected.

- *Trojan Horse and other code changes.* We used our system to detect changes in the behavior of FTP, after inserting a few lines of code. The change modified the location of most instructions, even those that corresponded to unchanged portions of code. Consequently, almost every system call made by the modified server was from a different program location as compared to the original server, thus leading to a continuous stream of anomalies. Note that, unlike other approaches that do not use program counter information, the FSA approach can detect changes to code even before the changed portions are executed.

- *Maliciously crafted input.* Several attacks rely on inadequate checking done by programs on their input data. By suitably altering the input (or command-line argument) an attacker can cause the program to behave unexpectedly. Our approach will detect these attacks, since the attacks induce programs to execute unusual sections of code and/or result in unusual system call traces. As an example, we detected the site exec vulnerability in the FTP program.

- *Dictionary or Password guessing attacks.* These attacks do not cause new sections of code to be executed, but are characterized by repetitive execution of same code. Such attacks can be detected by maintaining frequency-of-execution information with the automata edges.

- *Denial-of-Service Attacks.* Most DOS attacks cause server programs to execute some sections of their code very frequently. The FSA algorithm can be expected to detect them using frequency-of-execution information.

Based on our classification [27] of attacks reported in the CERT database, we note that the above classes of attacks account for about half of all attacks reported by CERT over the past few years.

4.5.1 Attacks Not Detected

- *Attacks that involve system call argument values.* Some attacks (e.g., attacks involving files accessed via symbolic links) differ from normal program execution only in terms of system call arguments. FSA and N-gram algorithms cannot detect such attacks, since there are no changes to the system call sequences.

- *Attacks that do not change behavior of attacked program.* Some attacks exploit errors of omission in the attacked program, such as, race conditions, opening of files without appropriate safeguards and checks, leaving temporary files with critical information etc. Exploitations of these errors are accomplished using a different program from the one containing the error, and thus do not cause the "attacked program" to behave differently. A second class of attacks that do not change program behaviors are those that exploit system configuration errors (e.g., user writable password file) or protocol weaknesses (e.g, SYN-flooding), and do not cause programs to misbehave. All these attacks are outside the scope of FSA and N-gram approaches.

- *Certain classes of attacks launched with knowledge of the intrusion detection techniques being used.* We indicated earlier that almost all buffer overflow attacks and Trojan Horse programs can be detected by the FSA algorithm. However, armed with the knowledge of how the FSA-based intrusion detection approach works, it is possible to develop successful buffer overflow attacks, as well as Trojans.

5 Conclusions

In this paper, we presented a new technique for intrusion detection based on learning program behaviors. Our method captures program behaviors in terms of sequences of system calls. These sequences are represented using a finite-state automaton. Unlike previous approaches, the FSA approach does not limit either the number or length of system call sequences. (Even without such limits, our representation ensures that the size of FSA itself is bounded – in the worst case, its size is linear in the size of the program.) Moreover, it captures the looping and branching structures of a program in a natural way, enabling it to recognize variations of behaviors learnt during training. The presence of program state information enables the FSA approach to perform more accurate detection of execution of unusual sections of code. Its ability to focus on program behaviors (while ignoring library behaviors) contributes to shorter training periods and smaller storage requirements.

Our experimental results support the following conclusions about the FSA method.

- *FSA-learning algorithms converge quickly.* The length of training required is one of the most important criteria for judging an anomaly detection technique. Our experiments show that in absolute terms, FSA learning converges quickly. For FTP, NFS and HTTP server, learning was completed after the servers used up several minutes of CPU time.

- *False positive rate of the FSA algorithm is low.* In relative terms, the FSA algorithm produces much fewer false positives than the N-gram algorithm. Even the absolute values are on the low side, corresponding to a rate of 10^{-4} or less after a moderate period of training. (On our web server, this would correspond to about 5 false positives a day.) In reality, the actual false positives experienced will be significantly lower, since it is unlikely that isolated deviations from the FSA model will be reported as attacks.

- *Space and runtime overhead of FSA-learning is minimal.* Our experiments show that the space requirements for the FSA algorithm is low. Its runtime overhead is also low.

- *FSA approach is effective in detecting attacks.* Our experiments show that the FSA approach can detect a wide range of attacks.

Several further improvements to the method are still possible. One promising avenue is the incorporation of frequency information along with the transitions, so that we can detect and flag attacks that involve many transitions that are associated with low probability of occurrence. Such an approach can detect many denial of service attacks.

A second avenue is the incorporation of system call argument values into the FSA. This extension will expand the set of attacks detectable by the approach to include many filename related attacks, such as those involving symbolic links.

References

[1] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, SRI-CSL-95-07, SRI International, 1995.

[2] CERT Coordination Center Advisories, `http://www.cert.org/advisories/index.html`.

[3] C. Cowan et al, StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, 7th USENIX Security Symposium, 1998.

[4] D. Endler, Intrusion Detection: Applying machine learning to solaris audit data, In Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC98).

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, Intrusion Detection using Sequences of System Calls, Journal of Computer Security Vol. 6 (1998) pg 151-180.

[6] T. Fraser, L. Badger, M. Feldman Hardening, COTS software with Generic Software Wrappers, Symposium on Security and Privacy, 1999.

[7] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson, SLIC: An Extensibility System for Commodity Operating Systems, USENIX Annual Technical Conference, 1998.

[8] A.K. Ghosh and A. Schwartzbard, A Study in Using Neural Networks for Anomaly and Misuse Detection, USENIX Security Symposium, 1999.

[9] A.K. Ghosh, A. Schwartzbard and M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, 1st USENIX Workshop on Intrusion Detection and Network Monitoring, 1999.

[10] A.K. Ghosh, A. Schwartzbard and M. Schatz, Using Program Behavior Profiles for Intrusion Detection, in Proceedings of the SANS Third Conference and Workshop on Intrusion Detection and Response, 1999.

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

[12] K. Ilgun, R. Kemmerer, and P. Porras, State Transition Analysis: A Rule-Based Intrusion Detection Approach, IEEE Transactions on Software Engineering, March 1995.

[13] K. Jain and R. Sekar, User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement, ISOC Network and Distributed Security Symposium, 2000.

[14] M. Kearns and L. Valiant, Cryptographic Limitations on Learning Boolean Formulae and Finite Automata, ACM STOC, 1989.

[15] C. Ko, G. Fink and K. Levitt, Automated detection of vulnerabilities in privileged programs by execution monitoring, Computer Security Application Conference, 1994.

[16] A. Kosoresow and S. Hofmeyr, Intrusion detection via system call traces, IEEE Software '97.

[17] S. Kumar and E. Spafford, A Pattern-Matching Model for Intrusion Detection, Nat'l Computer Security Conference, 1994.

[18] C. Michael and A. Ghosh, Using Finite Automate to Mine Execution Data for Intrusion Detection: A preliminary Report, Lecture Notes in Computer Science (1907), RAID 2000.

[19] T. Mitchem, R. Lu, R. O'Brien, Using Kernel Hypervisors to Secure Applications, Annual Computer Security Application Conference, December 1997.

[20] W. Lee and S. Stolfo, Data Mining Approaches for Intrusion Detection, USENIX Security Symposium, 1998.

[21] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman, Evaluating Intrusion Detection Systems: the 1998 DARPA Off-Line Intrusion Detection Evaluation, in Proceedings of the DARPA Information Survivability Conference and Exposition, 2000.

[22] T. Lunt et al, A Real-Time Intrusion Detection Expert System (IDES) - Final Report, SRI-CSL-92-05, SRI International, 1992.

[23] P. Porras and R. Kemmerer, Penetration State Transition Analysis: A Rule based Intrusion Detection Approach, Eighth Annual Computer Security Applications Conference, 1992.

[24] P. A. Porras and P. G. 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.

[25] L. Pitt and M. Warmuth, The minimum consistency DFA problem cannot be approximated within any polynomial, ACM STOC, 1989.

[26] L. Rabiner, A tutorial on Hidden Markov Models and selected applications in speech recognition, Proceedings of the IEEE, 1989.

[27] R. Sekar and Y. Cai, Classification of CERT/CC Advisories from 1993 to 1998, `http://seclab.cs.sunysb.edu/sekar/papers/cert.htm`

[28] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, USENIX Security Symposium, 1999.

[29] G. Vigna and R. A. Kemmerer, Netstat: A network-based intrusion detection approach, In Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98), pages 25-34, Los Alamitos, CA, December 1998, IEEE Computer Society, IEEE Computer Society Press. Scottsdale, AZ.

[30] C. Warrender, S. Forrest, B. Pearlmutter, Detecting Intrusions Using System Calls: Alternative Data Models, 1999 IEEE Symposium on Security and Privacy, May 9-12, 1999.