Finite Automata Models for Anomaly Detection

V. R. Ramezani[1]
Institute for Systems Research
The University of Maryland
College Park, Maryland 20742
e-mail: rvahid@isr.umd.edu

S. Yang[1]
Department of Electrical and Computer Engineering
Institute for Systems Research
The University of Maryland
College Park, Maryland 20742
e-mail: syang@isr.umd.edu

J. S. Baras[1]
Department of Electrical and Computer Engineering
Institute for Systems Research
The University of Maryland
College Park, Maryland 20742
e-mail: baras@isr.umd.edu

Abstract — **A fundamental problem in intrusion detection is the fusion of dependent information sequences. In this paper, we consider the fusion of two such sequences, namely the sequences of system calls and the values of the instruction pointer. We introduce FAAD, a finite automaton representation defined for the product alphabet of the two sequences where dependencies are implicitly taken into account by a matching procedure. Our learning algorithm captures these dependencies through the application of certain parameterized functions. Through the choice of thresholds and inner product structures, we are able to produce a compact representation of the normal behavior of a program.**

I. Introduction

Intrusion detection methods can be divided into two categories: misuse detection and anomaly detection. Misuse detection methods rely on detailed descriptions of attack signatures to detect known attacks. They are very effective against known attacks but not against new ones. In some cases, even minor variations of known attacks can be missed.

Anomaly detection is based on an approximate representation of normal behavior of the system. A behavior which significantly deviates from this normal representation is flagged as an intrusion. We present a new technique for program-based anomaly detection. For a partial list of anomaly detection methods see [2]-[7] and the references therein.

Forrest [2] and her collaborators have shown that the sequence of system calls can be used to represent the normal behavior of a program. Learning this behavior is accomplished by the use of the "N-grams" which are strings of length N observed during the normal use of the program. The N-grams characterize the pattern of system calls which occur during the normal execution of the program and are tabulated during the training phase. During the detection phase, sequences of system calls containing N-grams which deviate too much from the table of N-grams are considered anomalous. This approach is inspired by the intrinsic ability of biological immune systems to distinguish between "self" (the organism) and pathogens. There are some obvious limitations to this method: the long range behavior of the system calls and hence the program is not captured and the number of the N-grams grows exponentially with N. The number $N = 6$ has been suggested [2]; this number may indeed be justified but might depend on the data set itself [9].

[1]This work was supported by the U.S. Army Research Office under Award No. DAAD 190110494.

Finite Automata (FA) or Finite State Machines (FSM) have been suggested as an alternative to N-grams. (Throughout the paper, we will use the terms Finite State Machines and Finite Automata interchangeably.) First, a finite automaton can represent an infinite number of sequences of arbitrary length with a specific structure. Furthermore, once a finite automaton is constructed, combinations of all allowed transitions can generalize it to strings not used in the learning phase but "similar" to the ones used. Figure 1 shows this *path combination* property; suppose that we have recorded a transition ab from State 1 to 3 via 2 and also a transition cd for the same states. Then, the machine also accepts transitions ad and cb.

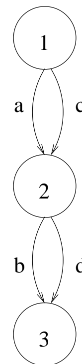

Fig. 1: path combination

In general, learning the structure of an FA from the set of its traces is a hard problem [10][8]. Sekar*et al* have introduced a novel method for the construction of an FA which uses the Instruction Pointer's value to determine the state and the sequence of the system calls to represent the transitions. The method shows promise in decreasing the rate of false alarms.

We see two draw backs with this method. First, the size of the automaton (the number of states) is not managed by the algorithm but is determined entirely by the distinct values of the instruction pointer (IP) over all training runs which could become too large (though in the examples presented the number of the states converges reasonably fast). The second draw back is their choice of states. Clearly, there are many variables which represent the behavior of the program other than the value of the IP. Thus, both of these values (IP and system calls) should be viewed as the manifestation of more abstract states.

We will present an algorithm in which:

1) The number of states is indirectly controlled by the parameters of the algorithm.

2) The states are abstract and the transition alphabet is

the product alphabet of the system calls and the values of the IP.

Forrest's N-gram method and the construction introduced by Sekar are both based on the alphabet of system calls alone and Sekar uses the values of the instruction pointer only to "label" the states. We believe the underlying alphabet must be a product of system calls and other relevant variables such as the values of the IP. We hope to extend this product alphabet to other variables and parameters. As expected, the construction resulting from the product alphabet of system calls and IP values is more general than a construction based on the language of system calls alone. We will show that one can view Sekar's machine as a special case of FAAD.

II. From the source code to an FA

In this section, we provide some insight as to how one can begin with the source code of a program and achieve an FA representation for the language of system calls whose states are labeled by the values of the IP. This essentially justifies Saker's approach.

At the atomic level, a program consists of a sequence of machine instructions. There are different types of instructions which might modify the control stream of a program:
-Regular instructions (increment IP and affect other states)
-Calls (push the IP and jump to another location)
-Return (pop the IP and jump back to original call location)
-Unconditional jumps (directly modify IP)
-Conditional jumps(either directly modify or increment IP)
-System calls

Note that in all programs, calls and returns are used in pairs to mark functions and the conditional jumps depend on the data stream as well as the control stream. The value of the IP represents a "location" in the memory in which the instruction to be executed next resides. We can label these locations with the values of the IP. As the program is executed a transition is made from one location to another. If a system call is made, we label the transition with that system call; otherwise, the transition is labeled with ϵ. We can then construct a non-deterministic finite automaton [11] by considering the locations labeled with the IP values as enumerating the states and by labeling the transitions with system calls or the null-character ϵ just as described above. The language of system calls accepted by this ϵ non-deterministic finite automata (ϵ-NFA) is regular and a superset of the language accepted by the program. This is caused by conditional jumps which depend on the data stream (which we do not observe) as well the control stream.

This ϵ-NFA can be reduced to an NFA by a standard procedure [11]. The NFA may be converted to a deterministic FA by observing the values of the IP as well as the system calls during a transition. It is not difficult to see that given all possible traces of a program, Sekar's construction will produce the above FA. In Sekar's method transitions are labeled by system calls and the states with IP values. But to be consistent with the definition of an FA, the transitions should be labeled by elements from the *product* alphabet of system calls and IP values. We will make this discussion precise in the next section.

We are not interested in all possible executions of the program and do not require or assume the knowledge of the source code. We define as "normal" those traces which are typically generated by the users of our system. It is this behavior that we would like to learn.

III. The Algorithm

As mentioned above, our alphabet is given by the product $P \times S$ where P is the set of values of the instruction pointer and S is the set of all possible system calls. We are interested in constructing an FA which accepts all training sequences which are traces of pairs of system calls and IP values. Formally, our algorithm constructs an FA given by

$$A = (Q, \Sigma, \delta, q_0, F),$$

where
1. Q is a finite set of states.
2. Σ is the set $P \times S$ defined above.
3. F is the set of accepting states; in our construction F=Q.
4. δ the transition function, defined from $Q \times P \times S$ to Q.

- The algorithm begins with an initial state. Given an initial machine, upon receiving an IP and system call pair, we create a new state to which a transition is made labeled with that element of the alphabet (the received pair of IP and system call), unless a state already exists with a transition directed to it whose IP component matches the IP component of the received pair of IP and system call. In that case, we will make a transition to that state and label it with the received pair rather than create a new one. This process continues until $2l$ states are constructed.

- From that point on, a forward window of length l is used to determine whether a new state should be created or an old one can be used upon the next transition (the new element of the training sequence). All the existing states will be considered as candidate states.

  -If a transition *out of* the present state labeled with the newly received element already exists, we take it. If one does not exist:

  -We examine each candidate state and test our training elements against the set of all $l+1$ possible transitions attached to the candidate state. The first one must be a transition *to* the candidate state and the remaining l must be transitions *from* that state and going forward. These are tested against $l+1$ elements of our alphabet starting at (and including) the new element and the next l elements of the training sequence. A categorical function will make the comparison and will return a real number r. This number is tested against a threshold value $s > 0$. The state with the largest $r \geq s$ is chosen for the transition. If the r value for all possible transitions in all states fails to meet the threshold, a new state is added and chosen for the new transition.

IV. The categorical functions

The two sets of $l+1$ elements described above are compared. Recall that each element is given by a pair; the comparison for each pair returns a pair (i, j) where $i, j \in \{0, 1\}$, 0 for a mismatch and 1 for a match. The $2(l+1)$ elements will form the vector $V = (i_0, j_0, ...i_l, j_l)$. Then $f(V) =: < V, W >$, where $W \in R^{2(l+1)}$ and $< \cdot, \cdot >$ is the inner product function. Different choices of W and s will give rise to different construction. In particular, we will prove that if $W_1 \geq s$ and $\sum_{i \neq 1} W_i < s$ the algorithm will reduce to Sekar's machine. Thus, Sekar's construction is a special case of our algorithm.

Theorem 1: If $W_1 \geq s$ and $\sum_{i \neq 1} W_i < s$, The algorithm described above reduces to Sekar's construction.

Proof:

The proof is by induction. It is not difficult to show that the initialization step is equivalent to Sekar's construction for the first $2l$ states.

We show that given the equivalence of the two construction for a number of states, say N, the algorithm will maintain the equivalence after the next iteration. Suppose $W_1 \geq s$ and $\sum_{i \neq 1} W_i < s$. Thus, $< V, W > \geq s$ iff $V_1 \neq 0$. This means that a new state is created only if no existing sate has a transition to it labeled with the value of the instruction pointer appearing in the next training pair. If so, then a new state is created with the new transition. Otherwise, an existing state is chosen to which a transition must exist with the same instruction pointer value (as in the next training element). This process does not allow for two transitions to the same state having different values of the instruction pointer. Thus, each state can be labeled with the value of the instruction pointer unique to it. And, the labeling of a transition to each state depends only on the value of the system calls. But, this is exactly how Sekar's machine is incremented.

Different choices of s, W and l will give rise to different machines. For example, consider the choice $l = 1$, $W =< 1, 0, 1, 0 >$ and $s = 1$. This causes that part of the algorithm which adds a new state to do so only if starting from an existing state, *two* consecutive instruction pointer values do not match and the choice $l = 1$, $W =< 0, 1, 0, 1 >$ and $s = 1$ will add a new state if *two* consecutive system calls cannot be matched. A choice such as $l = 1$, $W =< 0.5, 0.5, 0.5, 0.5 >$ and $s = 1$ considers combinations of *both* the system calls and the values of the instruction pointer. Allowing both system calls and the instruction pointer values to dictate the construction of states through the look forward window l leads to different representation of the program.

The instruction pointer value (assuming the program is executed in a single thread) represents the address of the instruction being executed. Consider again the example $l = 1$, $W =< 1, 0, 1, 0 >$ and $s = 1$. Roughly speaking this choice "merges" the two different *branches* of the program execution if they diverge only in one instruction. On the other hand, the choice $l = 1$, $W =< 1, 1, 0, 0 >$ and $s = 2$ may create two different states for the *same* value of the instruction pointer because the system calls in two different runs of the program may differ.

To see how the algorithm works, consider an existing machine displayed in Figure 2. Suppose, we have $W = < 1, 1, 1, 1 >$, $l + 1 = 2$, $s = 4$ and we are given a sequence $(a, i), (c, k), (b, h), (e, v)$..... We begin at State 0 and reach State 2 through State 1 and then must execute (b, h). There is no transition from State 2 labeled with (b, h). Thus, the machine has to evolve and either add a new state or use an existing one. Each state is considered as a possible transition. Consider State 3 and State 4.

-State 3 will give us a comparison between the path $(b, h), (e, v)$ vs. $(b, h), (e, v)$ and produces the inner product value $< V, W >=< (1, 1, 1, 1), (1, 1, 1, 1) >= 4 \geq s$.

-State 4 gives us a comparison between the path $(b, h), (e, v)$ vs. $(b, h), (e, u)$ and $< V, W >=< (1, 1, 1, 0), (1, 1, 1, 1) >= 3 < s$.

Examining all the other states shows that State 3 is the best choice and meets the threshold requirements. Thus, a new transition is added from State 2 to State 3 and labeled with (b, h).

Suppose now the transition from state 3 to 2 was labeled with a pair other than (e, v) then none of the states would have met the criterion $< V, W >\geq s$ and a new state would have been added and labeled with the transition (e, v).

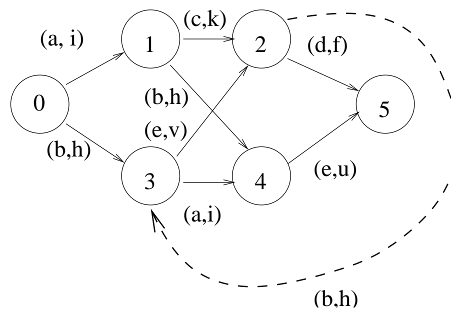

Fig. 2: Using an existing state in FAAD.

The anomaly detection phase is fairly simple. We examine a trace by putting it through the machine. Each time there is a mismatch, all states are considered as a *forward candidate state*. The forward window l as in FAAD is used to measure the number of matches and mismatches. Then the state with the least number of mismatches is chosen. The process continues a fixed number of trials x. If after x number of trials the machine did not accept the trace, then the trace is declared an anomaly. In this paper, we set $x = 1$.

Suppose a trace declared anomalous by FAAD, upon "expert" examination, turns out to be normal. We can then train the machine using this trace and assure that the machine would accept the trace. It is clear that the rate of false positives is non-increasing with training for FAAD. An attractive feature of FAAD is that training can continue after its use for intrusion detection begins.

V. Test Setup and Performance of FAAD

The test host is a Linux machine running a version of Washington University's FTP server (wu-ftpd) that has several well known vulnerabilities. These include a remote string format attack that can be exploited to gain root and a misconfiguration of the automatic tar conversion feature that might allow execution of arbitrary shellscripts by a remote user.

The training data consists of sequences of system call name and instruction pointer stack trace pairs. As noted in [1], the value of the instruction pointer that the system call is issued from is likely to be a location in glibc, which is uninformative. The actual instruction pointer containing useful information comes from the server code itself. So in this work, we use the entire instruction pointer stack trace in addition to the system call. To capture this information from the running ftp server, we use our own tracing tools based on the Linux ptrace facility.

There are two distinct parts of the data. A scripted ftp client that calls ftp services according to a Markov chain generates the normal data. The probability of calling a particular ftp function depends only on the previous ftp function called. Examples of ftp functions are put, get, status, and quit. We collected a total of 1,157,000 system calls comprising 876 individual execution traces of normal data. To ensure that the

testing data was realistic, we included in the test set data generated by interfacing with the ftp server manually with several commonly used ftp clients.

The attacks tested represent three different categories. The first category is the *string format attack*. We expect attacks from this category to be easily detectable because code is forced onto the stack and typically an exec system call is made from the stack, which would instantly be detected by our machine because no normal execution contains systems calls executed from the stack.

It is important to consider attacks that are not so easily detectable. The second category of attack is *misconfiguration*. For this, we executed an exploit of the automatic tar conversion [12]. This attack should not be easily detectable because it is a normal execution of the program that passes a maliciously crafted argument to tar.

Lastly, we construct a novel *stealthy attack*. This is based on the well known ftp bounce attack. In the bounce attack, the ftp port command is used to direct traffic towards a host other than the connected client. It can be used to make the ftp server conduct port scans behind firewalls. Since this became a well known attack, it was repaired by the Washington University development team. They disabled port forwarding to machines other than the originating host. Our novel attack works by exploiting the same string format attack as described previously to directly modify control state inside the ftp server. The effect of this attack is different from the string format to root exploit, where it is necessary to make a system call from the stack. This attack executes code from the stack but issues no system calls from the stack, so it should not be as easily detected.

As mentioned in the previous section, our algorithm depends on the length of the look forward window l, the threshold s and the choice of the inner product W. Table 1 compares the performance of FAAD vs Sekar's algorithm, where W is the choice of the inner product, s is the threshold, $|Q|$ is number of the states, $|L|$ is the number of links, FP is the number of False positives and DC the detection.

By Theorem 1, the first line of the table corresponds to Sekar's construction. The smallest machines are given by $W =< (1,1),(1,1) >$, $s = 1$ and $W =< (1,1) >$, $s = 1$; the first has about 1/5 and the second has about 1/4 as many states as Sekar's machine. The latter performs as well Sekar's algorithm. The former, despite its better compression, causes an increase in the rate of false positives from 1 in 177 to 2 in 177. One can interpret the choice of $W =< (1,1) >$ as taking into account the matching of both the IP values and the system calls with equal weights. Apparently this allows for a much smaller machine which performs as well Sekar's. There were 1002 different distinct transitions in the training set and thus the number of the links is always more than 1002 as expected.

If we only consider the IP values for matching, we can still reduce the size of the machine but we have to increase the size of the look forward window; with the choice $W =< (1,0),(1,0), (1,0),(1,0),(1,0) >$ and $s = 1$, we can reduce the size by about 300 hundred states without affecting performance. These results and our further tests suggest both the IP values and system calls should be taken into account.

Finally, as shown in the figures, the learning algorithm converges fairly fast in all cases after about 100 runs.

W	s	|Q|	|L|	F	D
$<(1,0)>$	1	1003	1257	1	3/3
$<(1,0)(1,0)>$	1	866	1120	2	3/3
$<(1,0)(1,0)(1,0)>$	1	809	1126	2	3/3
$<(1,0)(1,0)(1,0)>$	2	1041	1215	1	3/3
$<(1,0)(1,0)(1,0)(1,0)(1,0)>$	1	749	1135	1	3/3
$<(1,1)>$	1	254	1200	1	3/3
$<(1,1)(1,1)>$	2	204	1157	2	3/3
$<(1,1)(1,1)>$	3	1047	1248	2	3/3
$<(1,1)(1,1)(1,1)>$	3	368	1181	1	3/3
$<(1,1)(1,1)(1,1)>$	4	1017	1213	1	3/3
$<(1,1)(1,1)(1,1)>$	5	1098	1285	2	3/3

Tab. 1: Performance of FAAD

VI. Conclusions

We introduced an automaton based anomaly detection method. Our algorithm, Finite Automata Models for Anomaly Detection (FAAD), created a finite automata representation of the normal execution of a program. The structure of this automaton was learned from the runs of the program during its normal use. The number of the states was controlled and we did not require access to the program source code. FAAD captured typical programming structures such as loops and branches but without the excessive details of previous methods. Unlike most anomaly detection methods which rely on the alphabet of system calls, our method used the product alphabet of system calls and instruction pointers. One could consider more than two dependent information sequences. In principle, our learning algorithm could be applied to the product alphabet of these sequences through the appropriate choices of thresholds and inner product structures.

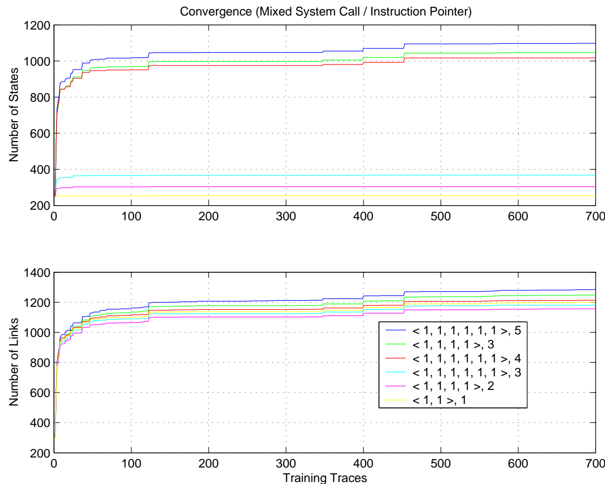

Fig. 3: Convergence of FAAD.

References

[1] Sekar et al. "A Fast Automaton-based Method for Detecting Anomalous Program Behaviors," *IEEE Symposium on Security and Privacy,* 2001.

[2] S. Forrest, S. A. Hofmeyr, A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *Journal of Computer Security,* Vol. 6 pp. 151-180, 1998.

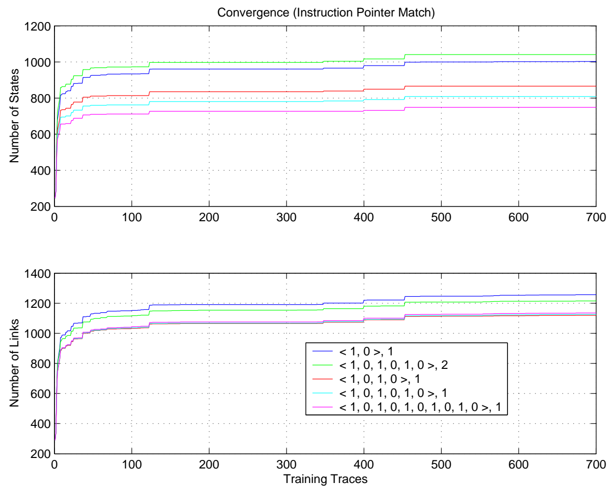

Fig. 4: Convergence of FAAD.

[3] 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.

[4] W. Lee and S. Stolfo, "Data Mining Approaches for Intrusions Detection," *USENIX Security Symposium,* 1998.

[5] P.A. Porras and P.G. Neumann, Emerald: "Event monitoring enabling responses to anomalous live disturbances," *In Proceedings of 20th National Information Systems Security Conferences,* po. 353-365, Oct. 1997.

[6] A. k. Ghosh and A. Schwartzbard, "A study in Using Neural Networks for Anomaly and Misuse Detection," *USENIX Security Symposium,* 1999.

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

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

[9] Carla Marceau (private communication) "Characterizing the Behavior of a Program Using Multiple-Length *N*-grams, Technical Report," *Odyssey Research Associates*

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

[11] J. E. Hopcroft, R. Motwani, J. D. Ulman, *Introduction to Automata Theory, Languages, and computation-2nd ed.* Addison Wesley, 2001.

[12] FTP Conversions on Misconfigured Systems, December, 1999 [Online]. Available: http://www.securiteam.com/