Dynamic Learning of Automata from the Call Stack Log for Anomaly Detection

Zhen Liu and Susan M. Bridges
Department of Computer Science and Engineering
Mississippi State University
{zliu, bridges}@cse.msstate.edu

Abstract

Anomaly detection based on monitoring of sequences of system calls has proved to be an effective approach for detection of previously unknown attacks on programs. This paper describes a new model for profiling normal program behavior that can be used to detect intrusions that change application execution flow. The model (hybrid push down automaton, HPDA) incorporates call stack information and can be learned by dynamic analysis of training data captured from the call stack log. The learning algorithm uses call stack information maintained by the program to build a finite state automaton. When compared to other approaches including VtPath which also uses call stack information, the HPDA model produces a more compact and general representation of control flow, handles recursion naturally, can be learned with less training data, and has a lower false positive rate when used for anomaly detection. In addition, dynamic learning can also be used to supplement a model acquired from static analysis.

1. Introduction

A recent research focus in the information assurance community has been monitoring of program behavior for detection of intrusions. This type of monitoring has proved to be an effective method for detecting attacks such as buffer overflow, Trojan horse, and format string that are often destructive and difficult to detect with other monitoring methods [1–3, 5–9, 12]. The behavior of a program is defined exactly by the binary executable and thus it should be possible to build a model of normal behavior based on the executable that can be used to detect deviations from normal behavior. Such a model should produce no false alarms and should not miss attacks that mimic normal behavior [9].

The key question is how to characterize the normal behavior of a program. Two basic approaches have been used to build models of system behavior based on system calls. The first method, introduced by Forrest, et al. [3] and studied by many others, uses audit histories to "learn" a model of normal sequences of system calls. The second approach, introduced by Wagner and Dean [12], derives a model of legal sequences of system calls from the program code using static analysis. Wagner [12] and Gao et al. [4] have described the difficulties inherent in static analysis. In this paper, we focus on dynamic learning of program behavior in which a dynamic learning method is used to learn a profile by analyzing the call stack from normal runs of a program.

We present an automaton model (HPDA) that includes call stack information. The incorporation of call stack information results in a model that is more powerful than those that only consider the sequences of system calls and thus this type of model is able to detect additional attacks as described by Feng et al. [2]. Feng et al. have described the advantages of *VtPath* in terms of detection ability and false positive rate over other methods. Our model is similar to the *VtPath* model of Feng et al. [2], but is capable of developing a more general model of program control flow. This paper describes a dynamic learning algorithm for the HPDA model and presents results comparing the performance of the model for anomaly detection with other approaches. Our model is similar to the *VPStatic* model proposed by Feng et al. [1]. However, *VPStatic* is learned by static analysis of program code in a manner similar to that of Wagner [12] and our model is acquired dynamically from an audit of the call stack log. Experimental results show that our model learns with less training data and has a lower false positive rate than *VtPath* [2] and *n-gram* [3] methods. In addition, our model can be acquired using a combination of static and dynamic learning. The focus of this paper is on the dynamic learning algorithm.

The remainder of the paper is organized as follows. Section 2 briefly reviews related research. Section 3 describes the hybrid push down automaton (HPDA) model developed in this research. A dynamic learning algorithm that can be used to acquire the model from audit data is described in Section 4. Section 5 presents experimental results and compares our approaches with others. Section 6 summarizes and discusses future work.

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)
0-7695-2315-3/05 $ 20.00 **IEEE**

2. Related work

System calls provide a rich resource of information about the behavior of a program. The *stide* algorithm of Forrest, Hofmeyr, and Somayaji [3] builds a profile of normal behavior by enumerating all unique, contiguous sequences of a predetermined, fixed length *n* that occur in the training data. Both the learning and detection processes for this method are very fast. The disadvantage of enumerating sequences is its lack of generalization. Many researchers have extended Forrest's *n-gram* approach by applying machine learning algorithms to acquire the profile from *n-gram* sequences instead of merely enumerating them. These methods include neural networks [5], the RIPPER rule-based classifier [8], non-deterministic finite state automaton derived from *n-grams* [9], and Hidden Markov Models [13].

In a different approach, Wagner [12] proposed modeling program behavior using static analysis. In Wagner's research, a finite state automaton (FSA) or push down automaton (PDA) is built by analyzing the source code of the program. The states of the automaton implicitly indicate the program counter (PC) of the program and one transition is associated with one system call or ϵ move. Giffin, Jha, and Miller [6, 7] have extended this method to learn the model from binary code instead of source code and solve the problem of tracking function call context by executable editing.

Researchers have shown that the call stack provides an additional rich source of information that can be used for detecting intrusions. Sekar et al. [10] used the program counter in the call stack and the system call number to create a finite state automaton. Their automata are learned from past audit data. Feng et al. [2] extended Sekar's approach to create the *VtPath* between two system calls. The *VtPath* is the difference between the call stack for two consecutive system calls that represents the exit and entry of function calls during the execution between the two system calls. Recently, Feng et al. [1] have proposed *VPStatic*, a deterministic push down automaton (DPDA) model that takes advantage of call stack information but is learned by static analysis of the binary executable.

3. HPDA model

A finite state automata (FSA) model provides an effective representation of the control flow inherent in the program executable. Wagner et al. [12] have shown that finite state automaton learned from the source code of a program can capture all possible execution paths. However, attackers can escape from detection by using the system call sequences that are accepted by the FSA but are not valid in a program execution. This problem is called the impossible path problem [2, 6, 12]. A push down automaton (PDA) model is able to eliminate the impossible path problem by using an extra stack that maintains the context between function calls [12]. However, the simulation of a PDA model is too computationally expensive to be used for real-time intrusion detection [6, 7, 12]. In this paper, we present a new model, the hybrid push down automaton (HPDA). This model retains the advantages of the PDA model but can be simulated more efficiently during detection and is able to detect new attacks. Instead of maintaining an extra stack like the PDA model of Wagner and Dean [12], the HPDA utilizes the call stack information maintained by the system. Our model is similar to the recently proposed *VPStatic* model [1] which also uses call stack information in the automaton. However, our model is learned by dynamic analysis and does not maintain an internal stack. Moreover, our model is more compact than *VPStatic* model.

3.1. Definition of HPDA

We propose a new model which we call the hybrid push down automaton (HPDA) that retains the advantages of the PDA model, but can be simulated more efficiently during detection and is able to detect new attacks. Instead of maintaining an extra stack like the PDA model of Wagner and Dean [12], the HPDA utilizes the call stack information maintained by the system. Figure 1 gives the source code for an example program and the corresponding HPDA model. Whenever a system call is invoked, the intrusion detection system is activated and checks if there is a transition that can move the current state to an allowable state with the current system call. Our model is an extension of the FSA model with the inclusion of system call stack information and is therefore a PDA which uses the system call stack rather than maintaining its own stack.

An HPDA is a 5-tuple: (S, Σ, T, s, A) where S is a finite set of states and Σ is a finite set of input symbols. In our model the set of input symbols is defined as $\Sigma = (Y \times Addr) \cup \epsilon$ where $Y = \{Entry, Exit, Syscall\}$ and $Addr$ is the set of all possible addresses defined by the program executable. The inputs are of four types.

1. *Entry* is a function call entry point. The associated address is the return address for this function.
2. *Exit* is a function call exit point. The associated address is also the return address for this function.
3. *Syscall* is the invocation point of a system call, where the associated address is the return address for this system call.
4. ϵ is the empty string.

In addition, T is the set of transition functions $(S \times \Sigma \rightarrow S)$, s is the start state $(s \in S)$, and A is the set of accept states $(A \subseteq S)$.

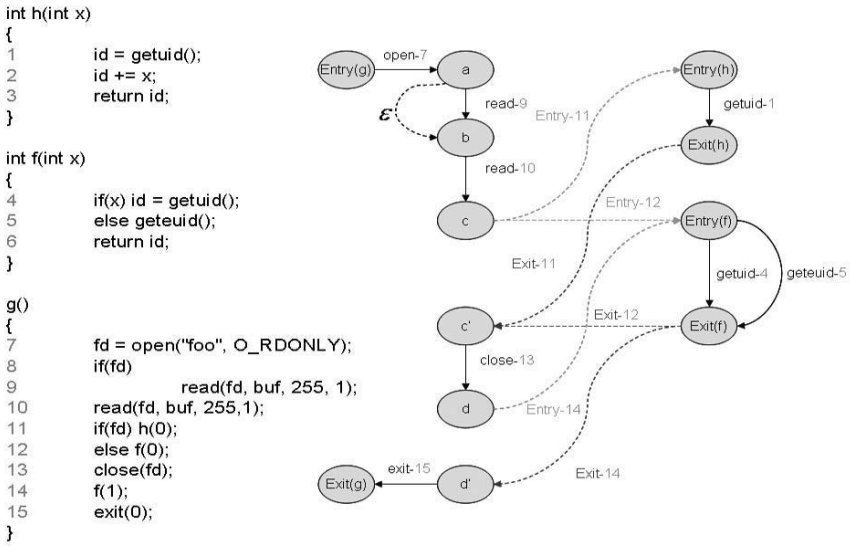

Figure 1. Example code and HPDA model

We assume that an address can uniquely identify an instruction. Therefore, the model is not applicable for programs utilizing overlay or self-modifying code that load different instructions into the same address. However, such techniques are seldom used in modern applications. Dynamic libraries can be loaded at different addresses during different executions and thus instructions in different dynamic libraries could be loaded into the same address during different runs of a program. However, the pair consisting of the relative address within the executable image (i.e. program executable, dynamic library executable) and the identity of the executable image can uniquely identify an instruction. In the remainder of this paper, *addr* denotes a pair consisting of a relative address (in either the program executable or a dynamic library executable) and the identity of the executable. This uniqueness property contributes to several important properties of the HPDA model that will be used by the dynamic learning algorithm.

3.2. Properties of HPDA

1. For each *Entry-Addr* in Σ, there is a corresponding *Exit-Addr* in Σ with the same address value.
2. All symbols in Σ have a unique address value except the paired *Entry-Addr* and *Exit-Addr* symbols.
3. All symbols in Σ can appear once and only once in any transition.

Although the definition of the HPDA appears to be similar to that of an FSA, it should be noted that the addresses that are incorporated into the input strings come from the system maintained stack. This gives our model the same capability as Wagner's PDA [12] and thus the name, hybrid push down automaton (HPDA).

4. Dynamic learning of an accurate model

In this section, we present an algorithm that dynamically learns the transitions in the HPDA model. With sufficient training data representing all possible normal behavior, the learned HPDA model for the example program will be the same as that presented in Figure 1. This dynamic learning algorithm can effectively capture the control flow and context of function calls. We obtain the call stack information maintained by a program and, in a manner similar to that used by Feng et al. [2], compute the difference between the call stacks associated with two consecutive system call invocations. The difference is then used by the algorithm to generate the transitions of the HPDA.

4.1. Algorithm

The differences between two system call stacks are computed and used for our system to learn new transitions in the HPDA model. The last call stack is CS_{last}, the current call stack is $CS_{current}$ and the current system call number is $SyscallID$. Each transition has a $from$ and to state. At some point during the program execution, the HPDA is in state S, and we get a system call $SyscallID$, $CS_{last} = (a_0, a_1, \ldots, a_l, a_{l+1}, \ldots, a_m)$, $CS_{current} = (b_0, b_1, \ldots, b_l, b_{l+1}, \ldots, b_n)$ where $a_0 = b_0, a_1 = b_1, \ldots, a_l = b_l, a_{l+1} \neq b_{l+1}, a_m$ is the return

address of last system call and b_n is the return address of current system call.

We need to create $Exit$ transitions associated with addresses $(a_{l+1} \ldots a_{m-1})$, *Entry* transitions associated with addresses $(b_{l+1} \ldots b_{n-1})$, and a *Syscall* transition with symbol *SyscallID-*b_n. Here we give the algorithm for $Exit$ transitions $(a_{l+1} \ldots a_{m-1})$. The algorithms for $Entry$ and $Syscall$ transitions are easily obtained by replacing the transition type in the algorithm. We examine the items in the sequence $(a_{l+1} \ldots a_{m-1})$ from a_{m-1} to a_{l+1}. Assume the address we are currently examining is a_i and the current state is S where $l < i < m$. Since the function *Exit* sequence is examined in the order a_{m-1} to a_{l+1}, the next address to be examined is a_{i-1}. In the case of *Entry* transitions the addresses are examined in the order b_{l+1} to b_{n-1}, the next address is b_{i+1} where $l < i < n$.

1. If there is an *Exit* transition out of *S* with address *Exit-addr* equal to a_i, change the current state to the state the *Exit* transition points to and try a_{i-1}.
2. If there is not an *Exit* transition associated with the current address from S, determine if there is an *Exit* transition in the entire diagram associated with a_i.
   a. If there is an *Exit* transition associated with an address equal to a_i in the diagram.
      (1) If both *S* and the *from* state of that *Exit* transition have an *out* transition, then an ϵ transition is added from *S* to the *from* state of that *Exit* transition. Change *S* to the to state of that *Exit* transition and try a_{i-1}.
      (2) If not, combine *S* with the *from* state of that *Exit* transition. Then change *S* to the *to* state of that *Exit* transition and try a_{i-1}.
   b. If there is no such transition in the entire diagram, create a new state, and create a new *Exit* transition from *S* to the new state with address a_i. Then change current state to the new state and try a_{i-1}.

The algorithm above considers two cases. In case 1, since a transition exists for the current state, it has already been learned. The only action is to change the current state for operating the HPDA. In case 2.a, there is a transition with the current input symbol. Based on property 3 of the HPDA, a new transition associated with the current symbol should not be created. Therefore the states need to be manipulated to allow both to use this transition. In case 2.a.(1), since both states have at least one *out* transition, they cannot be combined without compromising the constraints of the model. Therefore, an ϵ transition is created. In case 2.a.(2), the two states can be combined. Note that the state that does not have an *out* transition is always S, since the *from* state of the transition has been learned before must have an *out* transition. Therefore, S is merged with the *from* state of the transition. In case 2.b, there is no appropriate transition in the diagram. A new transition must be created with the current symbol. A new state is also created as the *to* state of the new transition.

The dynamically learned HPDA model can be treated as an alternative representation of *VtPath* proposed in [2]. Both approaches represent the call stack difference between two consecutive invocations of system calls. However, the HPDA representation is more compact, more general, and provides a more accurate representation of control flow than the *VtPath* representation. *VtPaths* are represented in a string form and saved in a hash table. One symbol appearing in several *VtPaths* needs to be saved several times. However, in the HPDA, each symbol appears in only one transition.

The HPDA also has an advantage when handling recursive function calls. In the *VtPath* model [2], recursion needs to be detected and treated as a special case, increasing the detection overhead. In the PDA model [12], the size of the extra stack for the PDA will increase dramatically when recursion is encountered and thus increase the memory and computational overhead. Sine the HPDA does not maintain a separate call stack and the structure of the automaton graph represents the recursive function relationship, no additional overhead is required to deal with a recursive function call.

Since *VtPath* treats the execution path as a string and records the strings in a hash table, it does not generalize the program control flow. Our dynamic learning algorithm can effectively capture the control flow structure and thus exhibits faster convergence during learning. For example, consider the program shown in Figure 1. After learning two traces that use *f(1)-14* twice (one called *getuid-4* and the other called *geteuid-5*), the flow structure in function *f()* has been learned. After learning a trace which contains a path *f(0)* and *getuid-4* the transitions *Entry-12* and *Exit-12* are learned. From this information, the HPDA model can infer that a path *Entry-12*, *geteuid-5*, *Exit-12* is valid in the program execution. However, *VtPath* will not accept this behavior until it encounters a trace that contains an execution path of *Entry-12*, *geteuid-5*, *Exit-12*. In section 5, experimental results demonstrate that the HPDA method has a lower false positive rate and faster learning rate than *VtPath*.

4.2. Online detection using HPDA

Our system is activated when a program invokes a system call. The HPDA is simulated using the symbols generated by computing the differences in the call stack to determine if the current invocation of the system call is allowed. The input symbols for the HPDA are generated from the difference between CS_{last} and $CS_{current}$. In this example, they are:

$$(\textit{Exit-a}_{m-1}, \ldots, \textit{Exit-a}_{l+1}, \textit{Entry-b}_{l+1}, \ldots, \textit{Entry-b}_{n-1}, \textit{SyscallID-b}_{n})$$

Then the HPDA is used to determine if there is a valid transition for each input symbol starting from the current state as follows.

> $S_{current}$ is the current state
> $T(s)$ contains the symbols that are associated with all transitions leaving $S_{current}$
> I is the ordered list of the input symbols generated from the difference between CS_{last} and $CS_{current}$
> At the beginning i is equal to the first symbol in I
> do
> if $i \in T(S_{current})$
> $S_{current}$ is set to the destination state of the transition that accepts i
> else
> an anomaly is detected
> $i = next\ symbol\ in\ I$
> while i is not null

It is possible for dynamic learning to result in false alarms because all behavior may not be represented in the training data. A false alarm generated by the HPDA may lead to a continuous sequence of false alarms. For example, in Figure 1, suppose the ϵ transition between state a and b is not learned. When the HPDA is in state a and a system call *read-10* is invoked, a false alarm will be generated and a remains the current state. Therefore, all succeeding normal calls will generate alarms. The following mechanism is used to recover from an anomaly in a graceful fashion. When an anomaly occurs, the program will determine if this symbol is in the HPDA. If there is a transition associated with the symbol, there is at most one according to property 3 of the HPDA. The new current state will become the destination state of this transition. Therefore, subsequent normal symbols will not generate alarms.

5. Experiments

Experiments were conducted to compare the learning convergence rates and false positive rates of HPDA, *VtPath*, *n-gram* models with window sizes of 2 and 6. Three

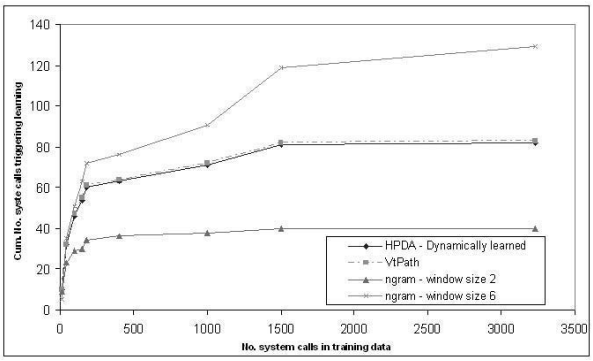

Figure 2. Convergence with *cat*

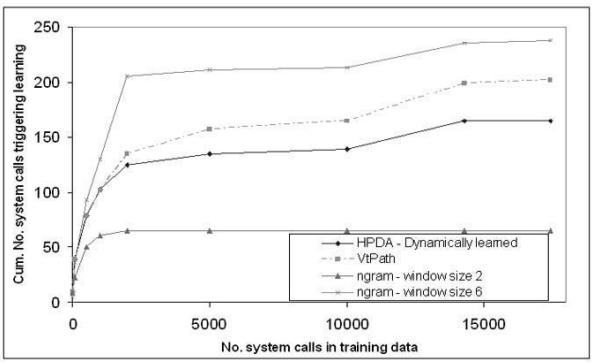

Figure 3. Convergence with *gzip*

programs that come with Red Hat Linux 9.0, *cat*, *gzip*, *vsftpd*, were chosen for the experiment. These programs allow demonstration of the performance of our method with programs ranging from the simple *cat* program to the complex ftp server *vsftpd*. Convergence speed is a measure of how fast a model can be learned by each method. The Y-axis in each of the figures is the cumulative number of system calls that trigger the model to learn. For *n-gram*, it is also the number of new sequences learned. For *VtPath*, it can be interpreted as the number of new paths learned. For HPDA, the learning consists of adding new transitions and/or states. The X axis is the number of system calls used in training.

In figures 2-4, the HPDA converges faster than *VtPath* and *n-gram* with a window size 6. The *n-gram* model with a window size 2 converges very fast but has been reported to have very poor detection capability [11].

Figure 5 presents the false positive rate of different methods with the program *vsftpd*. Both HPDA and *VtPath* have lower false positive rates than *n-gram* with a window size 6. The chart insert in figure 5 gives a closer view of the section of the large chart from 90-180 K system calls. From this section, we can see that the HPDA model gives a false posi-

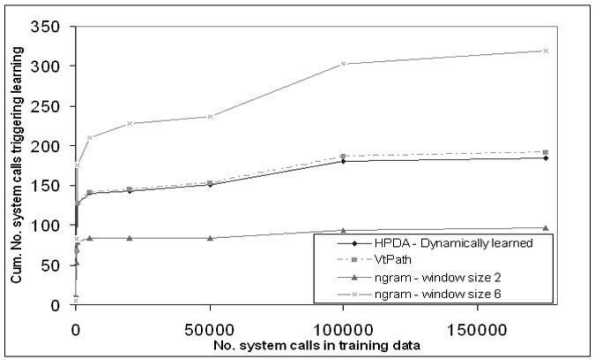

Figure 4. Convergence with *vsftpd*

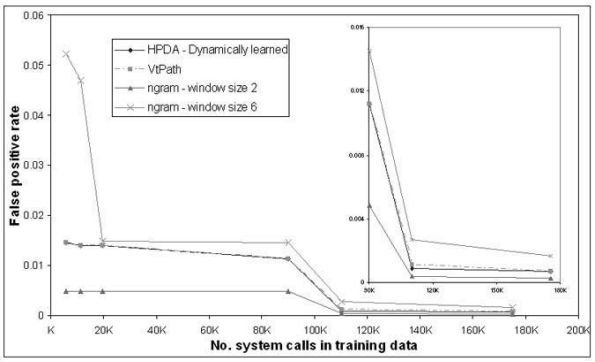

Figure 5. False positive rate with *vsftpd*

tive rate that is slightly lower than that of *VtPath*. Moreover, both HPDA and *VtPath* have false positive rates comparable to those of *n-gram* with a window size of 2 (no more than 0.0005).

6. Summary and future work

In this paper, we present a novel model, the HPDA, for modeling the behavior of programs. This model can be learned dynamically from audit data collected from the call stack log. The incorporation of information from the call stack into the model allows the HPDA to be used to detect more sophisticated attacks than methods which only consider the relative order between system calls. The automaton based structure of the HPDA yields a more general and compact representation of program control flow than *VtPath* and does not require that recursion be handled as a special case. Experiments show that the HPDA method can be acquired with less data and results in a lower false positive rate than *VtPath* and *n-gram* approaches.

The HPDA model can also easily be obtained by static analysis of the binary program executable. In other work, we have shown that dynamic learning algorithm can supplement a model acquired by static analysis in order to learn behavior that is defined at run-time.

References

[1] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings: IEEE Symposium on Security and Privacy*, Berkeley, California, 2004. IEEE Computer Society.

[2] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings: IEEE Symposium on Security and Privacy*, Berkeley, California, 2003. IEEE Computer Society.

[3] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings: IEEE Symposium on Security and Privacy*, pages 120–128, Los Alamitos, California, 1996.

[4] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings: USENIX Security Symposium*, 2004.

[5] A. K. Ghosh and A. Schwartzbard. Learning program behavior profiles for intrusion detection. In *Proceedings: 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, Santa Clara, California, 1999.

[6] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings: 11th USENIX Security Symposium*, 2002.

[7] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium*, 2004.

[8] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings: USENIX Security Symposium*, pages 79–94, San Antonio, Texas, 1998.

[9] C. Michael and A. Ghosh. Simple, state-based approaches to program-based anomaly detection. *ACM Transactions on Information and System Security*, 5(3):203–237, 2002.

[10] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings: IEEE Symposium on Security and Privacy*, pages 144–155, Oakland, California, 2001. IEEE Computer Society.

[11] K. M. Tan and R. A. Maxion. "why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings: IEEE Symposium on Security and Privacy*, pages 173–186, Berkeley, California, 2002. IEEE Computer Society.

[12] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings: IEEE Symposium on Security and Privacy*, pages 156–169, Oakland, California, 2001. IEEE Computer Society.

[13] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings: IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, California, 1999.