



PDF Download
2030376.2030377.pdf
19 December 2025
Total Citations: 3
Total Downloads: 234

 Latest updates: <https://dl.acm.org/doi/10.1145/2030376.2030377>

RESEARCH-ARTICLE

A gray-box DPDA-based intrusion detection technique using system-call monitoring

JAFAR HAADI H JAFARIAN, Sharif University of Technology, Tehran, Tehran, Iran

ALI ABBASI

SIAVASH SAFAEI SHEIKHABADI

Open Access Support provided by:

Sharif University of Technology

Published: 01 September 2011

[Citation in BibTeX format](#)

CEAS '11: The 8th Annual
Collaboration, Electronic messaging,
Anti-Abuse and Spam Conference
September 1 - 2, 2011
Perth, Australia

A Gray-Box DPDA-Based Intrusion Detection Technique Using System-Call Monitoring

Jafar Haadi Jafarian
Sharif Univ. of Technology
Tehran, Iran
jafarian@ce.sharif.edu

Ali Abbasi
Persian BugTraQ Group
Tehran, Iran
abbasi@bugtraq.ir

Siavash Safaei Sheikhabadi
safaei.siavash@gmail.com

ABSTRACT

In this paper, we present a novel technique for automatic and efficient intrusion detection based on learning program behaviors. Program behavior is captured in terms of issued system calls augmented with point-of-system-call information, and is modeled according to an efficient deterministic pushdown automaton (DPDA). The frequency of visit of each state is captured and statistically analyzed to detect abnormal execution patterns. This approach provides a very accurate learning of program behavior, which avoids a broad class of impossible path exploits. It also allows detection of new classes of attacks such as denial-of-service and brute-force dictionary attacks. We also present a complexity analysis of our model, and show that its time and space complexity is polynomial and fairly comparable to other similar approaches in learning, and hugely better in detection. Moreover, We evaluate our approach experimentally in terms of false positive rate, convergence rate, and performance. Finally, We shall discuss classes of attacks which are detectable and undetectable by our approach.

1. INTRODUCTION

Various classes of attacks on software systems cause abnormal behavior in process execution. Major examples include shellcode injection attacks on processes, resulting from vulnerabilities such as buffer overflow and format string vulnerabilities, and denial-of-service attacks. A significant amount of research has focused on detecting such attacks through monitoring behavior of the process and comparing that behavior to a model of normal behavior. These techniques are also called anomaly detection techniques because, in contrast to signature-based detection, deviations from the normal behavior are taken as indications of intrusions. The behavior that is considered in many recent research approaches is the sequence of system calls made by the process. This is mainly because system calls are the gateway between user processes and the operating system kernel, and a process is presumably able to affect its surroundings primarily through

system calls.

The use of automaton in modeling program behavior has been largely studied in the literature [1], [2], [3], [4], [5]. However, the approach presented here introduces new ways of constructing the normal behavior model, and offers properties for host-based anomaly detection that were not offered in prior techniques. Our approach builds a compact DPDA at runtime, in a fully automatic and efficient manner, without the need to access to source code for programs. Some advantages of runtime behavior learning for anomaly detection are described in [3] and [4]. Furthermore, our approach offers several innovative properties: I) It models program behavior according to a deterministic pushdown automaton (DPDA), which allows construction of very fast and efficient behavior models, while avoiding the impossible path problem. Since each transition is deterministic, the efficiency is high and the method will not miss intrusions due to non-determinism. The *impossible path* addressed in this paper extends the definition suggested by [4] beyond function calls. II) Frequency with which each transition is visited in different runs is statistically analyzed to disallow illegal execution patterns. Based on this frequency, if a transition is repeatedly visited in an abnormal manner, i.e. far beyond the frequency observed during training, an anomaly is announced. This property allows detection of new classes of attacks including denial-of-service and dictionary attacks, as well as improving the accuracy of the normal behavior model. III) The model allows detection of a broad range of previously-unaddressed attack patterns, with efficient runtime and space overhead.

The rest of this paper is organized as follows. Section 2 describes the related research. In Section 3 the training and detection algorithms are explored. Section 4 discusses important implementation issues. In Section 5 the model is evaluated both computationally and experimentally. The convergence and false positive rates of the model as well as its performance and overheads are described and compared to similar approaches. Section 6 describes classes of the attacks that are detectable or undetectable by our approach, and presents four relevant attack scenarios. Section 7 concludes the paper.

2. RELATED WORK

Many host-based intrusion detection systems (e.g., [6], [7], [8], [2]) and related sandboxing and confinement systems (e.g., [9], [10]) monitor the system calls emitted by a process in order to detect deviations from a previously constructed model of system call behavior.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CEAS '11 September 1-2, 2011, Perth, Western Australia, Australia
Copyright 2011 ACM 978-1-4503-0788-8/11/09 ...\$10.00.

Gao et al. [11], [5] coarsely divide these systems into §black box, §gray box and §white box approaches, based on the information they use to build the model to which they compare system calls at run time. Black-box and gray-box methods build a model of system-call behavior by monitoring sample executions. Specifically, black-box detectors employ only the system call numbers that pass through the system call interface when system calls are made (e.g., [6], [12]), and a gray-box detector extracts additional runtime information from the process, e.g., by looking into the process’s memory (e.g., [4], [3]). White-box approaches obtain the model by statically analyzing the source code or binary (e.g., [13], [14], [15], [2]).

These approaches were pioneered by Forrest et al. [6], who demonstrated that effective intrusion detection techniques can be developed by learning normal program behavior, and detecting deviations from this norm. To this aim, they suggested the use of system call sequences to model program behavior [16]. Their approach, which is referred to as *N-gram*, characterizes normal program behavior in terms of sequences of system calls. Wespi et al. [17] extended this approach to permit variable-length patterns of system calls, an approach later known as *vargram*.

The use of finite automaton in modeling program behavior has been largely studied in the literature. In an early example, Kosoresow et al. used DFA induction to statically learn a FSA that recognized the language of the program traces [1]. Wagner et al. [2] proposed to statically generate a non-deterministic finite automaton, called *callgraph*, and a non-deterministic pushdown automaton, called *abstract stack*, from the global control-flow graph of the program. The pushdown automaton ensured that every sequence of operations to the program call stack during a normal application execution would be among the set of paths explored during the simulation. The automaton was then used to monitor the program execution online. However, in these works no algorithm is provided for FSA or PDA construction and instead, they rely on human insight and intuition to construct automaton states and transitions from sequences.

[18] described an algorithm for constructing finite-state automaton from strings, but their algorithm treats only strings of a finite length. Thus, their approach learns tree-structured automaton. The problem of learning tree automaton is computationally much simpler than a general FSA that contains cycles [3]. [19] studied four different algorithms for learning program behaviors. Particularly interesting is 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.

Sekar et al. [3] proposed to generate a compact deterministic FSA by using system call traces and the program counter. A FSA using program counters is a close representation of the true structure of the code, and as such it is able to model loops and branches effectively.

The VtPath model [4] augments the FSA approach with stack return addresses. Each transition of the FSA includes a list of all the return addresses. This is used to generate a virtual path between system calls which can then be additionally checked for anomalies. This additional information improves attack detection and reduces false positives, without incurring additional overhead.

In a similar approach, execution graphs were used to extend simple system call enumeration to a more structured representation [11]. In this approach, which we refer to as *execution graph*, the return address pointer is stored with the system call, and this information is used to reconstruct a graph similar to a control flow graph by simply observing the patterns of system calls in a running program. The paper proves that given a set of observed program behavior, the algorithm constructs an execution graph that is consistent with a control flow graph which is obtained through static analysis.

3. BEHAVIOR MODELING AND ANOMALY DETECTION

The program behavior is modeled using a deterministic pushdown automaton. As each system call is made, we obtain the system call number 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). The challenges of extracting the program are discussed in Section 4.

Each distinct value of the program counter corresponds to a different state of the DPDA. The system calls correspond to transitions in the DPDA. The specification of a program DPDA P is written as follows:

$$P = \langle sttSet, \Sigma, \Gamma, \delta, strtStt, strtPath, fnlSet \rangle$$

- *sttSet*: A finite set of states. Each state has a *pc* attribute which shows the program counter associated with it. The *pc* is represented by two elements: a global block index, and an offset.
- Σ : A set of input symbols, i.e. system calls.
- Γ : A finite stack alphabet.
- δ : The transition function.
- *strtStt*: The start state.
- *strtPath*: The stack start symbol.
- *fnlSet*: The set of final states.

3.1 Training Phase

In this section, we provide a gray-box algorithm based on which a DPDA is learnt using system calls issued by a program, each of which are associated with a point-of-system-call information. Frequency-of-visit of each state is learnt during training and used based on statistical approaches to detect frequency-based anomalous executions.

Deterministic pushdown automaton of a program is created by a training process and using *LearnDPDA* algorithm presented in 1. The construction process continues through many different runs of the program, with each run possibly adding more states and/or transitions. The algorithm receives as input a target process, creates a child process for the target process and allows it to run without interruption as long as it is modifying its own process state. Whenever, the target (i.e. child) process issues a system call *SC*, the request is intercepted by the algorithm and the target process is suspended. The program counter associated with the system call, *PC*, is extracted and the algorithm

TrainWithSyscall presented in 2 is invoked with the pair (SC, PC) as argument.

Assume the following sequences were observed for a program. Figure 1 shows the PDA which is constructed for these sequences.

$seq_1 : \langle (SC_1, 1), (SC_2, 3), (SC_3, 7), (SC_4, 8), (SC_5, 10) \rangle$
 $seq_2 : \langle (SC_1, 1), (SC_6, 5), (SC_3, 7), (SC_4, 8), (SC_7, 12) \rangle$
 $seq_3 : \langle (SC_1, 1), (SC_8, 13), (SC_9, 15), (SC_8, 13), (SC_9, 15), (SC_{10}, 17) \rangle$
 $seq_4 : \langle (SC_1, 1), (SC_8, 13), (SC_{11}, 19), (SC_{12}, 20) \rangle$

The logic behind algorithm 2 is that, each run of a program is actually a sequence of system calls which represents a walk from the start state of its DPDA to one of the final states. Irrespective of back edges (i.e. transitions to states which are ancestors of the current state in a depth-first traversal from the start state), if a state has transitions to two different states, the set of outgoing transitions to each state represents a unique branch, which must be distinguished using a unique identifier. Accordingly, each walk can be represented by an ordered set of such identifiers. In the example above, seq_2 is a walk in the DPDA of figure 1, which can be demonstrated by the set of path identifiers $\langle a, c, e \rangle$. Such an approach allows us to distinguish all previously-observed walks and detect abnormal walks during detection. As long as a state has transitions to only one other state, it does not need a new identifier and simply preserves and relays received identifiers. But, as soon as one or more transition to another state is added to it, the set of transitions to each of these states must have a distinct branch identifier, so that each can be uniquely identified. Transitions to ancestor states must be handled differently, because they take control flow of the program to a previously visited state, and therefore no new branch identifier is required.

Determining the frequency with which a state is visited during normal executions allows us to prevent abnormal execution patterns. In order to do so, in each run, frequency-of-visit (FoV) for each state is recorded. After all training runs are observed and the training period is finished, the expected value and standard deviation of FoVs for each state are calculated based on the vector of non-zero FoVs for that state. Note that the set of unique values in FoV vector of each state denotes its sample space. A discrete random variable defined on this sample space denotes the number of times a state is visited during a run. The probability mass function for the random variable computes the probability of each value by dividing the number of occurrence of this value in the FoV vector by the length of this vector.

First we must determine if the observed FoVs are too spread out to be modeled using statistical concepts. If in contrast to its expected value, the standard deviation of an FoV vector is large, it shows that the FoV of the corresponding state changes drastically from execution to execution, and imposing any limitation on it may result in false-positive anomaly detections. To this aim, we use relative standard deviation (RSD) which is often expressed in percent, and is the ratio of standard deviation to expected value. The acceptable range of RSD can only be determined through experimental analysis. Based on our observation, if RSD is higher than 60%, the data is too dispersed around the average, and no limitation is imposed on the FoV.

Calculation of expected value (μ) and standard deviation

(σ) for the random variable defined on set of unique values in FoV vector of a state allows us to determine the range of permissible FoVs for that state. Assuming an unknown distribution for an FoV vector, we can use Chebyshev's inequality to specify an acceptable interval for future FoVs. Based on Chebyshev's inequality ($P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$), the probability that an FoV is in the interval $(\mu - 6\sigma, \mu + 6\sigma)$ is around 97%. Moreover, if a normal distribution is assumed for a random variable, based on the 3-sigma rule, 99.7% of values lie in $(\mu - 3\sigma, \mu + 3\sigma)$. Based on our observation, when the number of executions is large enough, this assumption seems to hold for many FoV vectors. However, the accuracy of such assumption requires empirical and statistical study and is left to future work.

The brute-force method to calculate expected value and standard deviation would be to store all of the FoVs for each state in an FoV vector. This method requires huge space overhead, since a vector must be kept for each state. For each state we only need the expected value and standard deviation of this vector, and FoV values are not required. Therefore, we use a method based on moving average, according to which the expected value of X and X^2 is calculated as each new value x_{i+1} arrives and up to current point:

$$E_{i+1}(X) = \frac{x_{i+1} + iE_i(X)}{i+1}$$

$$E_{i+1}(X^2) = \frac{x_{i+1}^2 + iE_i(X^2)}{i+1}$$

After observing all training runs, σ is calculated:

$$\sigma = \sqrt{E_n(X^2) - E_n(X)^2}$$

where n shows the number of x_{is} .

The following modifications must be applied to *LearnDPDA* and *TrainWithSyscall* algorithm to handle FoV-related computations:

- Initially, in *LearnDPDA* current FoVs of all states are set to 0:

```
for all  $stt \in sttSet$  do
   $crntFoV[stt] \leftarrow 0$ 
end for
```

$crntFoV[stt]$ shows the frequency-of-visit of stt for current run. It is in fact a random variable defined on the set of unique values in FoV vector of stt , i.e. its sample space.

- Whenever a new state stt is introduced, its attributes are initialized using *InitState*.

```
InitState( $stt$ )
```

- Whenever a state stt is visited, its current FoV is incremented:

```
Increment  $crntFoV[stt]$ 
```

- In *LearnDPDA*, after current run is terminated, the relevant values are updated based on current FoVs for all states:

```
for all  $stt \in sttSet$  do
  if  $crntFoV[stt] \neq 0$  then
     $avg[stt] \leftarrow \frac{crntFoV[stt] + vstCnt[stt] * avg[stt]}{vstCnt[stt] + 1}$ 
     $ex2[stt] \leftarrow \frac{crntFoV[stt]^2 + vstCnt[stt] * ex2[stt]}{vstCnt[stt] + 1}$ 
    Increment  $vstCnt[stt]$ 
  end if
end for
```

$vstCnt[stt]$ represents the number of non-zero FoVs observed for stt during all runs. $avg[stt]$ calculates the

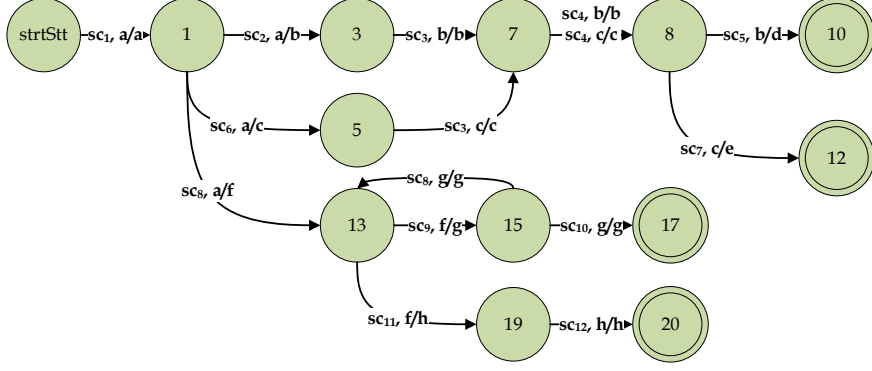


Figure 1: DPDA learnt by *LearnDPDA* algorithm.

moving average (i.e. expected value) for FoV vector of stt , and $stdv[stt]$ represents its standard deviation.

- Finally, after the training phase is over, the standard deviation and RSD is computed for each state. If for a state, RSD is higher than a threshold (60% in our implementation), no limitation is imposed on its FoV.

```

for all  $stt \in sttSet$  do
   $stdv[stt] \leftarrow \sqrt{ex2[stt] - avg[stt]^2}$ 
   $rsd[stt] \leftarrow \frac{stdv[stt]}{avg[stt]} * 100$ 
  if  $rsd[stt] > threshold$  then
     $avg[stt] \leftarrow \infty$  {no limitation on FoV}
  end if
end for

```

Algorithm 1 LearnDPDA

```

LearnDPDA(target process)
   $crntStt \leftarrow strtStt$ 
   $crntPath \leftarrow strtPath$ 
  execute target process and intercept system calls
  for each intercepted system call  $SC$  do
    extract  $PC$ 
     $TrainWithSyscall((SC, PC))$ 
  end for
  if ( $crntStt \notin fnlSet$ ) then
     $fnlSet \leftarrow fnlSet \cup crntStt$ 
  end if

```

In addition to abnormal execution patterns, the FoV information can also contribute to a better modeling of program behavior. Suppose that in a program the pseudo code 4 exists. In training run, the following sequence is observed for this program:

$\langle ..(SC_j, 1), (SC_i, 3), (SC_i, 3), (SC_k, 5).. \rangle$

In our approach, this will create a self-loop on the state accepting SC_i . Without FoV information, this automaton will accept sequences such as the following:

$\langle ..(SC_j, 1), (SC_i, 3), (SC_i, 3), (SC_i, 3), (SC_k, 5).. \rangle$

But our approach detects such anomalies because the expected value and standard deviation for this state will be equal to 2 and 0 respectively, and the FoV of this state must be exactly equal to 2.

Algorithm 2 TrainWithSyscall

```

TrainWithSyscall((SC, PC))
if there exists no  $nextStt \in sttSet$ , such that
   $PC[nextStt] = PC$  then
    create a new state  $nwStt$ 
     $PC[nwStt] \leftarrow PC$ ,  $sttSet \leftarrow sttSet \cup nwStt$ 
     $nextStt \leftarrow nwStt$ 
  end if
if there exists some  $\gamma, \eta \in \Gamma$  such that  $\delta(crntStt, SC, \gamma) = \{(nextStt, \eta)\}$  then
  if  $\gamma \neq crntPath$  then
    if  $\gamma \neq \eta$  then
       $\delta(crntStt, SC, crntPath) \leftarrow \{(nextStt, \eta)\}$ 
    else
       $\delta(crntStt, SC, crntPath) \leftarrow \{(nextStt, crntPath)\}$ 
    end if
  end if
   $crntPath \leftarrow \eta$ ,  $crntStt \leftarrow nextStt$ 
else
  if there exists some non-loop transition from  $crntStt$  to any state, and  $nextStt$  not ancestor of  $crntStt$  then
    for all  $stt \in sttSet$  such that there exists a transition from  $crntStt$  to  $stt$  do
       $nwPath \leftarrow$  an unused stack alphabet
      for all transitions  $trn$  such that  $trn = \delta(crntStt, SC, \gamma) = \{(stt, \gamma)\}$  do
        replace  $trn$  with  $\delta(crntStt, SC, \gamma) = \{(stt, nwPath)\}$ 
        do a DFS traversal from  $stt$ , ignoring all ancestor states and replace all occurrences of  $\gamma$  with  $nwPath$  in all transitions
      end for
    end for
     $nwPath \leftarrow$  an unused stack alphabet
     $\delta(crntStt, SC, crntPath) \leftarrow \{(nextStt, nwPath)\}$ 
     $crntPath \leftarrow nwPath$ 
  else
     $\delta(crntStt, SC, crntPath) \leftarrow \{(nextStt, crntPath)\}$ 
  end if
   $crntStt \leftarrow nextStt$ 
end if

```

Algorithm 3 InitState

```
InitState( $stt \in sttSet$ )  
   $vsCnt[stt] \leftarrow crntFoV[stt] \leftarrow avg[stt] \leftarrow ex2[stt] \leftarrow 0$   
  increment  $crntFoV[stt]$ 
```

Algorithm 4

```
1:  $SC_j$   
2: for  $l = 0$  to  $1$  do  
3:    $SC_i$   
4: end for  
5:  $SC_k$ 
```

3.2 Online Detection Phase

Anomaly detection in our approach is managed by *monitor*. The monitor is a process which creates a child process for the target process and allows it to run without interruption as long as it is modifying its own process state. Whenever, the target (i.e. child) process issues a system call, the request is intercepted by the monitor and the target process is suspended. Next, the current program counter is extracted and the validity of system call is checked based on the DPDA, and any observed abnormality is announced. The anomalies are handled using an algorithm named *HandleAnomaly*.

To ensure that isolated mismatches do not immediately result in an intrusion being flagged, a leaky bucket algorithm is typically used (as in [21]) to aggregate anomalies over time. Each time an anomaly is detected, *HandleAnomaly* increments an anomaly count based on the severity associated with the anomaly. When the anomaly count exceeds a threshold, an intrusion is flagged. Similarly, when the monitor observes a normal behavior, the anomaly count is hugely decreased. As a result, the leaky bucket emphasizes anomalies that are closely temporally co-located and diminishes the values of those that are sparsely located.

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. This weight is determined by the severity of anomaly. In general, four major classes of anomalies can be detected by the monitor:

- *Stack corruption anomaly*. If any problem occurs in retrieval of a program counter, it would be because the stack has been corrupted, possibly due to a stack corruption exploit such as a buffer-overflow or format string attack. This class of anomalies severely compromises confidentiality, integrity and availability of the program, and therefore its severity is set to *high*. 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 monitoring halts after occurrence of such anomalies.
- *Missing state anomaly*. This anomaly occurs when the extracted program counter for current system call does not match any state. This is either due to insufficient training, or occurrence of an attack which may have been the resulted in, say, shellcode injection or return address modification. The severity of this anomaly class is *medium*. Accordingly, the weight associated

with this class is smaller than stack-related anomalies, such that several successive occurrences of these anomalies must occur before the threshold for flagging an intrusion is reached. When the monitor confronts a missing state, it moves the DPDA to a sink state. From now on, for each intercepted system call the program counter is extracted, and 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 codes.

- *Missing transition anomaly*. This anomaly occurs when the expected state exists in the automaton, but there is no qualified transition to it from the current state. This again might be due to insufficient training, or attacks resulting in shellcode injection, code change, or return address modification. The severity of this class is also *medium*. For the sake of re-synchronization, whenever the monitor confronts a missing transition, it handles the anomaly, but moves PDA to the supposed state without changing the stack.
- *FoV anomaly*. If a state is visited more than the specified boundary, an FoV anomaly occurs. This again might be the result of insufficient training, or a denial-of-service or brute-force attack. The severity of this class of anomaly is *low*, and the weight assigned to it is smaller than other classes.

Algorithm 5 Monitor

```
Monitor(Target Process)  
  for all  $stt \in sttSet$  do  
     $crntFoV[stt] \leftarrow 0$   
  end for  
   $crntStt \leftarrow strtStt$   
   $crntPath \leftarrow strtPath$   
  execute and monitor target process and intercept system calls  
  for each intercepted system call  $SC$  do  
    extract  $PC$   
    if any problem occurs during PC extraction then  
       $HandleAnomaly(stack\ anomaly)$  {severity: high}  
      return false  
    else  
       $CheckSyscall((SC, PC))$   
      if anomaly count is greater than threshold then  
        return false  
      end if  
    end if  
  end for  
  return true
```

4. IMPLEMENTATION ISSUES

Several approaches have been proposed for system call tracing over the past several years. Some of these techniques involve modifying the operating system kernel, as in [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 [22], and [23] make use of

Algorithm 6 CheckSyscall

CheckSyscall((SC, PC))

```
if there exists a  $nextStt \in SttSet$ , such that  $PC[nextStt] = PC$  then
  if  $crntStt = sinkStt$  then
     $crntStt \leftarrow nextStt$ 
  else if there exists some  $\gamma \in \Gamma$  such that  $(nextStt, \gamma) \in \delta(crntStt, SC, crntPath)$  then
    Increment  $crntFoV[nextStt]$ 
    if  $avg[nextStt] \neq \infty$  and  $crntFoV[nextStt] > avg[nextStt] + 6 * stdv[nextStt]$  then
      HandleAnomaly(FoV anomaly) {severity: low}
    end if
     $crntStt \leftarrow nextStt$ 
     $crntPath \leftarrow \gamma$  {update stack accordingly, i.e. POP once and PUSH  $\gamma$  into stack}
  else
    HandleAnomaly(transition anomaly) {severity: medium}
     $crntStt \leftarrow nextStt$ 
  end if
else
  HandleAnomaly(state anomaly) {severity: medium}
   $crntStt \leftarrow sinkStt$ 
end if
```

the process tracing capability provided by most versions of UNIX-based operating systems in order to perform system call interception at the user level. We used the second approach in this work. The advantage of monitoring at user level is that training and detection can be performed via unprivileged user-space applications which do not need kernel privileges to intercept system calls and, therefore, do not increase the trusted computing base (TCB) for processes which are not running on top of it. Increasing the size of the TCB is detrimental to the overall security of a system [23], [24]. In our implementation, we use *ptrace* with *Ptrace_PEEKUSER* to intercept system calls of the target process for both training and detection.

Wagner et al. [2], Sekar et al. [3] and Feng et al. [4] addressed some implementation issues for implementing gray-box system-call-based anomaly detection approaches. The main issues include handling non-standard control flows, determining program counter in dynamically linked libraries, and handling threads in PDA learning.

In handling non-standard control flows, we used the approach adopted by Feng et al. in [4]. For signals, when the first system call in a signal handler is executed, the information about the last system call is saved, including its PC. When the signal handler returns, we restore the information about the last system call. This framework can be easily extended for the multi-level signal handler case. Each execution of signal handlers is treated like a program run. The same techniques used for training and online detection can still be applied here with signal handlers. In case of *setjmp()/longjmp()*, Wag et al. showed that, unlike static methods, handling *setjmp()/longjmp()* pair is straightforward in dynamic approaches [2].

4.1 Retrieval of Program Counter in DLLs

One problem witnessed in our method as well as other similar gray-box methods is related to determining the program

counter in dynamically linked libraries (DLLs). The main problem is that the functions within DLLs may be loaded at different relative locations (comparing to the static portion) for different program executions, so the program counters may change from execution to execution. The FSA method tried to solve this problem by traversing the stack back to the statically linked portion. In this method, initially the value of the program counter is examined to see if it is from the statically linked portion of the executable. If not, the top-most frame on the stack is examined, and the return address information is extracted. If this address again does not correspond to a statically linked region of the program, the next stack frame (corresponding to the next outer procedure invocation) is considered and the same process is repeated. This method oversimplifies the model for DLLs, because intrusions may also occur in the DLLs. For example, the Trojan version of a DLL might be installed by an attacker, and the approach adopted by FSA method in modeling the DLLs can not detect these intrusions. In modeling functions in DLLs, we use an approach similar to the method used by Feng et al. [4].

In our approach, DLLs are modeled like any statically linked function. In training, a lookup table is used to keep information for each executable memory block of each forked process. This information includes the block length, its full path name, and the offset of the file from which the memory block was loaded. This information is used to distinguish the same memory blocks in different executions. When a system call is intercepted, last return address is retrieved, and it is used along with the lookup table to determine its memory block and the relative offset within the block. These two pieces of information can uniquely distinguish a program counter. Each program counter is represented by a global block index, and an offset within the block. One problem that may occur during this process is when we can not match a memory block to any memory block which has occurred during training. This can be because the intruder is trying to load another DLL.

For linux, the pseudo file $\$maps$ under the process's information pseudo file system $\$/proc$, contains all required information for each block, including its virtual memory address range, full path of the file from which this block is loaded, offset in the file, device number for the file, etc. Using the above approach, we can recognize the same dynamically loaded code block during different runs, and uniquely distinguish a program counter, just like statically linked functions.

4.2 Thread Management

One of the major difficulties in implementation of system-call-based anomaly detection methods, is dealing with threads. Currently, there are many different ways to implement threads. In all these approaches, it is straightforward to determine that an intercepted system call belongs to which thread, and therefore our approach can be easily applied to multi-threaded applications.

For Linux, different threads actually have different process identifiers, so each thread can be identified by its unique ID. In our work we have dealt with fork and exec system calls. These system calls require special attention, because they may change the running process, which may result in a corresponding change in the DPDA associated with the process. The fork system call causes the process to create a copy of

itself. Unless followed by *execve*, the child process performs the same tasks as that of the parent, and the child DPDA will be similar to its parents'. In our approach, a global DPDA is used to capture the behavior of the parent and all its children. After the fork, subsequent system calls made by either the parent or the child is added as a transition to the same DPDA, and distinguished via its ID. This requires us to keep track of a current state for each process. When one of these processes make a system call, a transition is added to the current state of this process. Also, an additional attribute is added to each transition, which specifies the ID of the process that is allowed to take it. During detection, this attribute determines which processes are allowed to take which transitions.

When an *execve* system call is made, a different DPDA is created for the new program. During detection, whenever the *execve* system call is issued, the corresponding DPDA for the executed program is retrieved and used.

5. EVALUATION AND DISCUSSION

5.1 Computational Complexity Analysis

Since in our approach, when a new symbol is pushed, the previous symbol is popped first, the DPDA requires a stack of length one. So, our DPDA is in fact a deterministic FSA augmented with one variable, v . As a result, $\delta(p \in \text{sttSet}, \alpha \in \Sigma, \gamma \in \Gamma) = \{q \in \text{SttSet}, \eta \in \Gamma\}$ can be rewritten in terms of FSA in the following way:

if current state is p and $\delta(p, \alpha) = q$ and $v = \gamma$

change state to q and $v \leftarrow \eta$

This simply shows that the overhead of each transition in our DPDA is slightly greater than a transition in FSA and still bound to $\theta(1)$. This would have not been the case if we used a larger stack.

The worst-case outdegree of a directed graph with v nodes is $v - 1$. Therefore, the worst-case time complexity of checking whether there exists a qualified transition from one specific state to any other state is $O(v)$. The time complexity of finding a qualified state is $O(v)$, and the time complexity of determining whether a state is ancestor of another state is $O(e+v)$, i.e. the time complexity of a DFS traversal. The worst-case scenario is that for each state we need to perform a DFS traversal for all its children. Therefore the worst-case time complexity of *TrainWithSyscall* is $O(e.v + v^2)$. This algorithm is called for each intercepted system call of the target process. The time complexity for interception of a system call is $O(t)$ where t is the runtime overhead for system call interception, and determining the program counter. Therefore, the total cost for observing each system call is equal to $O(t + e.v + v^2)$. In practice, the runtime overhead is much lower, since it would rarely occur that a state has transitions to all other states, or we would need to perform a DFS for children of all states. Also, it is evident that the number of states are equal for DPDA and FSA, but the number of transitions is higher in DPDA.

In the FSA algorithm introduced in [3], this overhead is equal to $O(t + v)$, since determining the ancestor relationship, and also DFS traversals are not required. The space complexity of DPDA learning and storage is $O(e + v)$.

The worst-case time complexity of *CheckSyscall* for each system call is $O(v)$. Therefore the time complexity per each system call is $O(v + t)$. In the FSA algorithm, the detection time complexity for each system call is $O(v + t)$. This

overhead increases for techniques which require more runtime information during program execution [4], [5]. Also, the extraction method is very important. Based on our observation, less than 1% of the system calls are actually made in the statically linked portion. For more than 30% of the system calls, the FSA algorithm has to go back at least 3 call stack frames to find a return address in statically linked portion. This means that at least 3 levels of DLL functions are called for 30% of the system calls. We use a more efficient method in determining program counter of system calls which are invoked in DLLs, which is explained in 4.

5.2 Experimental Evaluation

To assess different aspects of our technique, we compile and test executables of standard UNIX ftp client (ftp-0.17-51) and an ftp server (*pure-FTPD 1.0.29*), which we refer to as *ftp* and *ftpd* respectively. *ftpd* has a complex behavior, supporting almost 70 different operations. *ftp* is moderately complex supporting about 30 operations. All performance evaluation are performed on an Intel Core 2 Duo Processor E8400 3.0 GHz system running Fedora Linux 12 and Linux Kernel 2.6.31-5. For these experiments, we used training scripts that generated commands to exercise the executables. Specifically, we developed a script generator program which mimics some common user activities, such as downloading/uploading files and directories. The program assured that all commands and options are executed with all types of inputs. For example, for *mkdir* command, it made sure that all types of arguments including illegal and existing directory paths were given. Also, almost all valid and invalid permutation of command sequences were generated. The training scripts were used to generate larger and larger sequences of commands in successive runs. While it would have been better to run all of the tests on live servers, such an approach was impractical because we did not have access to systems that experienced large enough volumes of traffic to enable us to conduct such experiments. In order to compare the results, we also implemented the PC-based FSA learning method proposed by Sekar et al. [3], and also used the evaluation results published in [3] and [4]. The program behavior observed during each run was learnt using the FSA and DPDA algorithms. The initial run included very few commands, typically resulting in about a thousand system calls made by the program. The final run included several million system calls.

5.2.1 Convergence

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.

The convergence rate is closely related to program size. For *ftp*, the DPDA converged before one million system calls, while for the *ftpd*, the algorithm converged around five million system calls. After convergence, the algorithm did not learn any thing new even when the number of system calls were increased by an order of magnitude.

As already expected, our approach has a slightly higher convergence rate than both FSA and VtPath methods, because a higher number of transitions is required to determine the acceptable system call paths. For example, in figure 1 two transitions are required between states 7 and 8, each representing a different path. Figures 2 and 3 show the con-

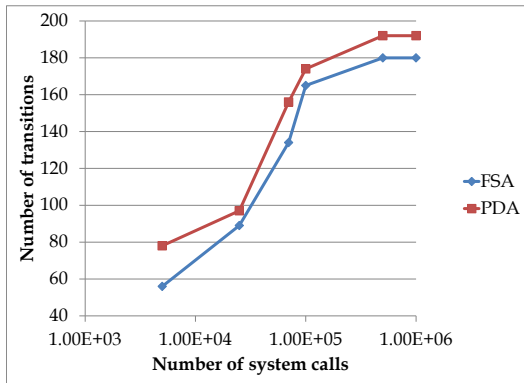


Figure 2: Convergence on *ftp*.

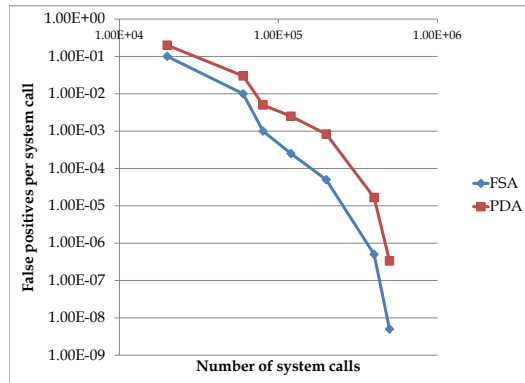


Figure 4: False positive rates on *ftp*.

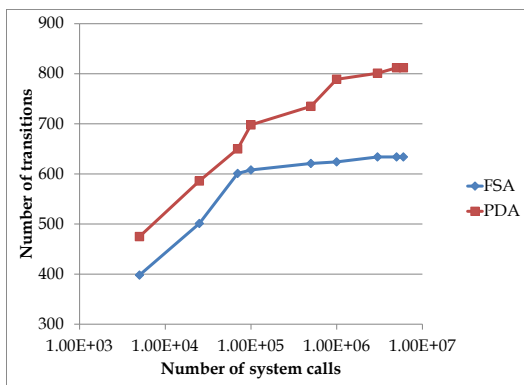


Figure 3: Convergence on *ftpd*.

vergence rate for *ftp* and *ftpd* using I) our implementation of FSA algorithm, and II) our DPDA algorithm. Feng et al. [4] compares FSA and VtPath algorithms in terms of convergence time. In these figures, the number of transitions are plotted against the numbers of system calls used for training. The graphs use a linear scale on the Y-axis and a logarithmic scale on the X-axis. In comparing the two algorithms, the actual Y-axis values are not important, what matters for convergence is whether the curves flatten out quickly. As stated in 5.1, for an equal training sequence, DPDA naturally requires more transitions than FSA, although the number of states are the same. For *ftp* the convergence rates are almost equal for both algorithms. For *ftpd*, DPDA has a slower convergence rate, although it becomes more similar to FSA after 10^6 system calls. In contrast to *ftp*, this slower convergence rate is due to complex behavior of *ftpd*. An important reason for higher convergence rate in FSA is partly related to its oversimplification in determining required program counter, which oversimplifies the DLL structure.

5.2.2 False Positives

To compute the false positive rate, we collected several normal testing traces ranging from several hundred to sev-

eral million system calls for each method, with a script execution distribution slightly different from what was used for the convergence experiments. This script was again generated by our own script generator. The exact same system call traces were used to train and analyze the FSA and DPDA algorithms. We used the profiles saved in the convergence experiments to analyze these testing traces. Each valid system call detected as an anomaly is counted as a false positive. The false positive rates are calculated as the number of false positives over the number of system calls in each trace, and averaged over the several testing traces for each method.

Figures 4 and 5 show the number of false positives reported by each algorithm for *ftp* and *ftpd* respectively. Both axes are in logarithmic scale. The false positive rates of the algorithms are closer for *ftp*. For *ftpd*, the DPDA algorithm has a higher false positive rate, which is mainly due to FoV-related information. As effective as it is, the FoV information may produce a great amount of false positives, specially for less trained samples. We believe that this false positive rate can be reduced substantially using more accurate statistical approaches. For *ftpd*, the false positive rate of our algorithm falls below 10^{-6} after a training period corresponding to about 10^6 .

5.2.3 Performance and Overheads

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 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. The reason for adoption of this approach is explained in 4.

We use the same user-level mechanism to intercept system calls as in previous methods. We use the average process time per system call stop to evaluate the algorithm runtime overhead. For FSA algorithm, this value is about 350 milliseconds for training and 250 milliseconds for detection. For VtPath, the value is announced to be about 150 milliseconds for both training and detection [4]. For our algorithm,

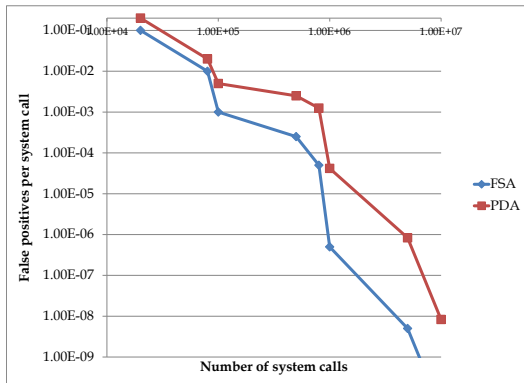


Figure 5: False positive rates on *ftpd*.

the average runtime overhead for each system call is around 400 milliseconds, and around 120 milliseconds for detection. This improvement in detection time is due to the fact that compared to the FSA method [3], our approach uses a more efficient technique in extraction of additional runtime information, and the amount of extracted information is less than the information required in methods in [4] and [5].

Implementation techniques and optimization may substantially reduce such overheads. For example, in our implementation, we used a transition reduction method which acts based on merging similar transitions. Similar transitions are defined as those transitions that have the same start and end states. In practice, adoption of this method results in optimization of our training runtime up to 50 percent.

The space overhead for DPDA algorithm is slightly greater than FSA. Also, we used a DPDA compression method, based on which states with indegree and outdegree of 1 are merged with previous states. This technique reduces the number of states and transitions to a great extent. For *ftpd* experiments, the profiles that the DPDA code creates are about 15K bytes, close to the FSA method. In this measurement, the FoV-related information is also included for DPDA.

6. ATTACK DETECTION

In this section, we describe classes of attacks that are detectable and undetectable by our approach, and also present four illuminating attack scenarios.

6.1 Detectable Attacks

In general, the following classes of attacks can be detected by our approach.

- **Memory error exploits.** The vast majority of today’s security vulnerabilities continue to be caused by memory errors, with a significant shift away from stack-smashing exploits to newer attacks such as heap overflows, and format string attacks. Buffer overflow attacks are still among the major software threats, and more than 500 buffer overflow-related exploits and vulnerabilities were reported in 2009 [25]. Buffer overflow refers to a class of vulnerabilities which emanates from failure to constrain operations within the bounds of a

memory buffer, either stack or heap. Buffer overflow exploits manipulate the program in one of the following ways: I) By overwriting a local variable that is near the buffer in memory to change the behavior of the program which may benefit the attacker. II) By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user input filled buffer. Attack Scenario II in section 6.3 exemplifies such exploits. One other primary example is *return-to-libc* attacks. III) By overwriting a function pointer, internal structures such as linked list pointers (in heap-based buffer overflow exploits), or exception handler, which is subsequently executed or retrieved. If the buffer overflow attack corrupts the stack in a way that the retrieval of program counter fails, the attack can be easily detected. In other cases, such attacks can again be detected by our approach, if it has an effect on system call sequences issued by the program. This effect can be either through invocation of a new system call, or by changing the control flow of the program. Otherwise, the attack goes undetected. Attack scenario III in Section 6.3 gives an example of such undetectable buffer overflow attacks. Another major class of memory errors are format string attacks. The format string vulnerability can be exploited to force a program to overwrite the address of a library function or the return address on the stack with a pointer to some malicious shellcode. This can be easily detected by our approach, unless the shellcode does not have any effect on system call sequences issued by the program.

- **Code changes.** If the code change introduces an unexpected system call sequence, or an expected system call at an unexpected location, it will be detected. This property allows detection of trojan horses, including trojan versions of dynamically loaded libraries.
- **Improper input validation.** 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. Fuzzing methods are usually exploited to discover such malicious inputs. These attacks can be easily detected, since it will induce programs to execute unusual sections of the code and/or result in unusual system call.
- **FoV-based attacks.** These attacks do not cause new sections of code to be executed, but are characterized by repetitive execution of the same code. Two main classes of such attacks are denial-of-service and brute-force dictionary attacks. This abnormal execution can be detected using the FoV information associated with each state. Attack scenario IV provides an example on FoV-based attacks.
- **Impossible path exploits.** Feng et al. restrict the impossible path problem to function calls. Based on their definition, if a function is called from one place but returns to another, an impossible path occurs. They solve this problem by keeping a list of all return addresses for each transition [4]. This definition can be extended to include all paths which will never

occur in a normal run. In this work, the DPDA properties are used to distinguish each path and disallow illegal system call sequences. The use of pushdown automaton for avoiding the impossible path problem was initially proposed by Wagner et al. in [2] in a completely different approach. Feng et al. showed that the monitor efficiency for this method is too low [4], which is mainly due to complexity of pushdown automaton and the non-determinism of the simulation. Our method excludes the element of non-determinism, and in Section 5 we show that our DPDA is in fact a deterministic FSA augmented with one auxiliary variable. Attack Scenario I in Section 6.3 illuminates our approach in the detection of impossible paths. The class of impossible paths covered by our approach is not only limited to function calls but also includes a broader range of such exploits.

- **Unexpected program behavior.** In general, any type of vulnerability exploit which may cause the target program to behave unexpectedly (with respect to system calls) during monitoring can be detected by our approach. Exploiting vulnerabilities such as race condition within threads, race condition in signals, divide by zero, integer overflow, type conversion errors, race condition in switch statement, weak error/exception handling, NULL pointer dereference, and memory leaks often result in unexpected program behavior which can be detected by our approach.

6.2 Undetectable Attacks

Here, we describe some of the attacks that can not be detected by our method.

- **Mimicry attacks.** Similar to other conventional system call monitors, our method is susceptible to mimicry attacks, e.g., [26]. Mimicry attacks can remain undetected by keeping system calls the same as those that would have been invoked by the legitimate program, while only changing some of the system call arguments. For example, assume a legitimate Apache server opens an HTML file and sends its contents over the network. A mimicry attack could keep the *open* system call intact and pass the path of a file that contains sensitive information instead of the HTML file to the system call. In this scenario the Apache server would send sensitive information over the network and a the system call monitor would not be able to detect the attack. The link following exploits such as UNIX hard link and UNIX symlink following usually belong to this category, unless they result in an expected behavior.
- **Attacks with no system call effect.** In general, attacks which have no effect on system call sequences and return addresses will evade all the approaches discussed here. The attack scenario III in Section 6.3 provides an example of such attacks. Such attacks exploit the fact that many anomaly-based IDS only check their program behavior models at the time of system call. This attack will not be detected by any of the approaches we described so far and our approach is not an exception.
- **External attacks.** Some attacks exploit errors of omission in the attacked program, such as, TOCTOU

race conditions, opening of files without appropriate safeguards and checks, leaving temporary files with critical information, etc. Exploitations of these errors are accomplished using an external program different 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.

6.3 Attack Scenarios

In this section, we present four attack scenarios and describe how our approach deals with them.

6.3.1 Attack Scenario I

Assume that the pseudo code in 7 represents a section of the program the DPDA of which is shown in figure 1. For this code, the previous approaches, including VtPath and execution graph consider sequences such as the following one as valid, while this sequence demonstrates an impossible path which should not occur in any run.

$\langle \dots, (SC_1, 1), (SC_3, 3), (SC_4, 7), (SC_5, 9), \dots \rangle$

If the attacker succeeds to incorporate some changes so that the second *if* condition has a reverse effect, the above sequence goes undetected by FSA, VtPath, and execution graph methods. Our solution avoids such impossible paths by distinguishing each separate branch with a separate identifier, and determining the set of branches that can be taken together in a certain path. As it is shown in figure 1, the above sequence is not considered valid in our DPDA.

Algorithm 7 Pseudo code for attack scenario I

```

1:  $SC_1$ 
2: if some condition then
3:   authenticate user ( $SC_2$ )
4: else
5:   reject user ( $SC_6$ )
6: end if
7:  $SC_3$ 
8:  $SC_4$ 
9: if user is authenticated then
10:  redirect to panel ( $SC_5$ )
11: else
12:  redirect to login ( $SC_7$ )
13: end if

```

6.3.2 Attack Scenario II

For pseudo code in 8, if SC_4 is part of a command which contains an overflow, the attacker can substitute the return address of the function so that the procedure *DoSth* returns to (III). This is an example of function-based impossible path problem, which is also detectable by VtPath and execution graph models. Since the VtPath model keeps a list of return addresses for each transition, this attack can be detected by this approach, but it goes undetected in all other approaches.

6.3.3 Attack Scenario III

For pseudo code in 8, if the attacker returns to anywhere between *DoSth* and the next system call, the attack goes

Algorithm 8 Pseudo code for attack scenarios II & III

```
1:  $SC_1$ 
2: if some condition then
3:   authenticate user ( $SC_2$ )
4: else
5:   reject user ( $SC_6$ )
6: end if
7:  $DoSth()$ 
8: (I) some important code, including no system call
9: (II) some code
10: if user is authenticated then
11:   switch to superuser mode ( $SC_5$ )
12: else
13:   (III) switch to unprivileged mode ( $SC_7$ )
14: end if
15: procedure  $DoSth()$ 
16:    $SC_3$ 
17:    $SC_4$  {Part of a command which contains an overflow}
```

undetected. For example, if $DoSth$ is returned to (II) after overflow exploit, the important codes in line (I) are not executed. But this attack goes undetected, because no illegal system call or program counter is observed, and the program does not deviate from the normal behavior.

Algorithm 9 Pseudo code for attack scenario IV

```
1: (I)  $SC_1$ 
2: if not  $AuthenticateUshr()$  then
3:   login prompt {goes to (I)}
4: end if
5:  $SC_2$ 
6: ....
7: procedure  $AuthenticateUshr()$ 
8:    $SC_3$  {get username}
9:    $SC_4$  {get password}
10:   some code for user authentication
```

6.3.4 Attack Scenario IV

The pseudo code in 9 shows a part of a program which authorizes user based on her credentials. In case the credentials are not valid, the user is prompted again. Figure 6 shows the DPDA of this code along with the FoV vectors of each state for 10 runs, expected value and standard deviation of each state, and the permissible range of FoVs for non-training runs. When the attacker launches a brute-force dictionary attack on this program, the states 1, 8, and 9 are going to be visited far more than training time. The permissible FoV for these three states must be less than or equal to 8, and the 9th try will be detected by the monitor.

7. CONCLUSION

In this paper, we presented a new technique for intrusion detection based on learning program behaviors. The behavior model is represented by a DPDA. Our technique completely avoids the impossible path problem, i.e. no previously unobserved path would be accepted in the detection phase. This definition is broader than the definition presented in previous works, in which only the function calls are considered. Moreover, the frequency-of-visit of each state is captured and statistically analyzed to detect abnormal

execution patterns. This property allows detection of new classes of attacks such as denial-of-service and brute-force attacks. We also presented an analytical evaluation of our model, and demonstrated that its time and space complexity is polynomial and fairly comparable to other similar approaches, and better in detection. Moreover, we evaluated our approach experimentally in terms of false positive rate, convergence rate, and performance. We also discussed the type of attacks that can and can not be detected and by our approach.

One other work which is left to future is to improve the statistical modeling of states using FoV information. Currently we assume an unknown distribution. But based on our observation, the FoV information of many states have a normal distribution, especially when the number of runs increase.

8. REFERENCES

- [1] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.
- [2] *Intrusion detection via static analysis*, 2001.
- [3] *A fast automaton-based method for detecting anomalous program behaviors*, 2001.
- [4] Henry H. Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 62+, Washington, DC, USA, 2003. IEEE Computer Society.
- [5] Debin Gao. Gray-box anomaly detection using system call monitoring, 2007.
- [6] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, 1996.
- [7] Niels Provos. Improving host security with system call policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, page 18, Berkeley, CA, USA, 2003. USENIX Association.
- [8] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical report, Berkeley, CA, USA, 1999.
- [9] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.
- [10] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.
- [11] Debin Gao, Michael K. Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, page 8, Berkeley, CA, USA, 2004. USENIX Association.
- [12] Kymie Tan, John Mchugh, and Kevin Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Information Hiding*, LNCS, pages 1–17. Springer, 2003.

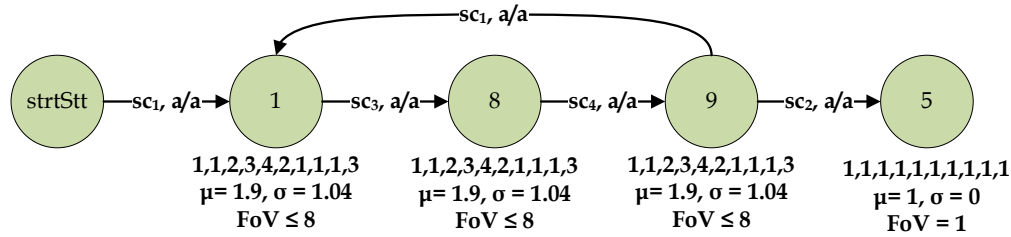


Figure 6: DPDA for pseudo code of attack scenario IV.

- [13] Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *In IEEE Symposium on Security and Privacy*, 2004.
- [14] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, Berkeley, CA, USA, 2002. USENIX Association.
- [15] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. 2004.
- [16] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.
- [17] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In *RAID '00: Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 110–129, London, UK, 2000. Springer-Verlag.
- [18] Christoph Michael and Anup K. Ghosh. Using finite automata to mine execution data for intrusion detection: A preliminary report. In *RAID '00: Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 66–79, London, UK, 2000. Springer-Verlag.
- [19] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *In IEEE Symposium on Security and Privacy*, pages 133–145. IEEE Computer Society, 1999.
- [20] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association.
- [21] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *ID'99: Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring*, pages 6–6, Berkeley, CA, USA, 1999. USENIX Association.
- [22] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *In Proc. Network and Distributed Systems Security Symposium*, 1999.
- [23] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46, New York, NY, USA, 2009. ACM.
- [24] Bernhard Kauer. Oslo: improving the security of trusted computing. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.
- [25] MITRE Common Vulnerabilities and Exposures Project.
- [26] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *ASIACCS*, pages 156–167. ACM, 2008.