Selecting and Improving System Call Models for Anomaly Detection

Alessandro Frossi, Federico Maggi, Gian Luigi Rizzo, and Stefano Zanero

Politecnico di Milano, Dipartimento di Elettronica e Informazione
{alessandro.frossi,gian.rizzo}@mail.polimi.it,
{fmaggi,zanero}@elet.polimi.it

Abstract. We propose a syscall-based anomaly detection system that incorporates both deterministic and stochastic models. We analyze in detail two alternative approaches for anomaly detection over system call sequences and arguments, and propose a number of modifications that significantly improve their performance. We begin by comparing them and analyzing their respective performance in terms of detection accuracy. Then, we outline their major shortcomings, and propose various changes in the models that can address them: we show how targeted modifications of their anomaly models, as opposed to the redesign of the global system, can noticeably improve the overall detection accuracy. Finally, the impact of these modifications are discussed by comparing the performance of the two original implementations with two modified versions complemented with our models.

Keywords: Anomaly Detection, System Call Models, Deterministic Models, Stochastic Models, Self Organizing Map.

1 Introduction

Since the seminal work of Forrest et al. [1], system call-based anomaly detection enjoyed immense popularity. The core of any anomaly detection system consists of a *composition* of effective *models* to *accurately* capture the observed system *behavior*.

While usually the approach is to re-design the whole system, we propose a much more effective way of improving over previous results. We selectively identify well-performing models, and compose them in novel ways to create improved detectors. To demonstrate our point, two alternative and quite *complementary* techniques [2,3] are chosen, in order to have a rich set of models to analyze and improve. In particular, we focus on incremental models improvements, and on cross-pollination among different approaches. We show how this process of analysis and improvement leads to globally improved detection accuracy with minimal efforts, as opposed to the re-design of the global system structure. We concentrate on the use of unsupervised learning algorithms, because this type of learning uses rather complex models and representations, creating an ideal testing ground for model improvement. Also, while most models are only based

U. Flegel and D. Bruschi (Eds.): DIMVA 2009, LNCS 5587, pp. 206–223, 2009.

on the program control flow, we deem it important to analyze also the content of the calls, as many attacks today are not exclusively based on control flow deviations.

The first prototype we analyze is based on a *Finite State Automaton* (FSA) augmented with dataflow information. We show that its promising capabilities (e.g., precise modeling of the control flow and solid relationship) are paid dearly in terms of low robustness. Indeed, several false detections are triggered by slight differences between the actual parameters and the learned, crisp models. On the opposite hand, we examine a model based on Markov chain modeling augmented by statistical anomaly models. It is able to capture frequency information and to infer relationships between different arguments of same system call, but has a number of shortcomings in terms of false positives and negatives.

We propose a set of modifications that can address some of the shortcomings of these prototypes. The impact of these modifications is analyzed by comparing performance and detection accuracy of the two original prototypes versus two modified, hybrid versions complemented with the new models. Without taking into account arguments values, hybrid systems based on both syscall sequences and control/data flows are not more accurate than pure control flow based ones [4]. On the other hand, we empirically show how the accuracy of a data flow IDS increases if call arguments are included in the models.

The remainder of this paper is organized as follows. In Section 2 we describe the two different prototypes implemented in previous works, along with the improvements we describe in Section 3. In Section 4 we evaluate the *Detection Rate* (DR), the *False Positive Rate* (FPR), and speed of the original and modified systems. In Section 5 we review the most relevant, recent host-based anomaly detection proposed in the literature.

2 Two Existing Approaches to System Call Anomaly Detection

In this section we describe the results of the analysis we conducted on the chosen anomaly detection systems.

2.1 FSA-Based Implementation

The first prototype we analyzed is a deterministic IDS which builds an FSA model of each monitored program [2], on top of which it creates a network of relations (or *properties*) among the system call *arguments* encountered during training. In the following, we call it "FSA-DF" as a shorthand. Such a network of properties is the main difference w.r.t. other FSA based IDSes. Instead of a pure *control flow* check, which focuses on the behavior of the software in terms of sequences of system calls, it also performs a so called *data flow* check on the internal variables of the program along their existing cycles.

This knowledge is exploited in terms of *unary* and *binary* relationships. For instance, if an `open` system call always uses the same filename at the same point,

```
1 int foo(char* dir, char* file) {
2   source_dir = dir; target_file = file;
3   out = open(target_file, WR);
4   push(source_dir);
5   while ((dir_name = pop()) != NULL) {
6     d = opendir(dir_name);
7     foreach (dir_entry ∈ d) {
8       if (isdirectory(dir_entry))
9         push(dir_entry);
10      else {
11        in = open(dir_entry, RD);
12        read(in, buf);
13        write(out, buf);
14        close(in);
15      }
16    }
17  }
18  close(out);
19  return 0;
20 }
```

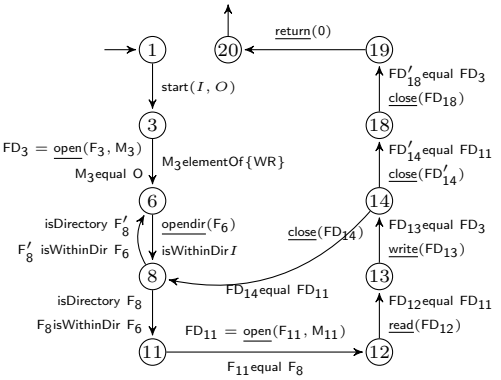

Fig. 1. A data flow example with both unary and binary relations

a unary property can be derived. Similarly, relationships among two arguments are supported, by inference over the observed sequences of system calls, creating constraints for the detection phase. Unary relationships include equal (the value of a given argument is always constant), elementOf (an argument can take a limited set of values), subsetOf (a generalization of elementOf, indicating that an argument can take multiple values, all of which drawn from a set), range (specifies boundaries for numeric arguments), isWithinDir (a file argument is always contained within a specified directory), hasExtension (file extensions). Binary relationships include: equal (equality between system call operands), isWithinDir (file located in a specified directory; contains is the opposite), hasSameDirAs, hasSameBaseAs, hasSameExtensionAs (two arguments have a common directory, base directory or extension, respectively).

The behavior of each application is logged by storing *Process IDentifier* (PID), *Program Counter* (PC), along with the system calls invoked, their arguments and returned value. The use of the PC to identify the states in the FSA stands out as an important difference from other approaches. The PC of each system call is determined through *stack unwinding* (i.e., going back through the activation records of the process stack until a valid PC is found). FSA-DF obviously handles process cloning and forking.

The learning algorithm is rather simple: each time a new value is found, it is checked against all the known values of the same type. Relations are inferred for each execution of the monitored program and then pruned on a "set intersection" basis. For instance, if relations R_1 and R_2 are learned from an execution trace T_1 but R_1 only is satisfied in trace T_2, the resulting model will not contain R_2. Such a process is obviously prone to false positives if the training phase is not exhaustive, because invalid relations would be kept instead of being discarded. Figure 1 shows an example (due to [2]) of the final result of this process. During detection, missing transitions or violations of properties are flagged as alerts. The detection engine keeps track of the execution over the learned FSA, comparing

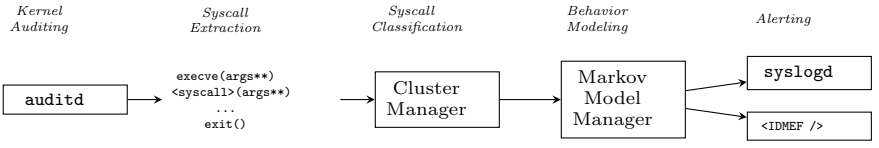

Fig. 2. The high-level architecture of our S^{2}A^{2}DE prototype

transitions and relations with what happens, and raising an alert if an edge is missing or a constraint is violated.

The FSA approach is promising and has interesting features especially in terms of detection capabilities. On the other hand, it only takes into account relationships between different types of arguments. Also, the set of properties is limited to pre-defined ones and totally deterministic. This leads to a possibly incomplete detection model potentially prone to false alerts. In Section 3 we detail how our approach improves the original FSA-DF implementation.

2.2 Markov Chains-Based Implementation

The second prototype we analyze is called S^{2}A^{2}DE (Syscall Sequence and Argument Anomaly Detection Engine) [3,5]. It exploits Markov chains to describe the behavior of a process. More specifically, S^{2}A^{2}DE analyzes processes as sequences of system calls $S = [s_1, s_2, s_3, \ldots]$. Each call s_i is characterized by a *type* (e.g. `read`, `write`, `exec`, etc.), a list of *arguments* (e.g., the resource path passed to `open`), a *return value*, and a *timestamp*. Neither the return value nor the *absolute* timestamp are taken into account.

S^{2}A^{2}DE can be decomposed in the basic blocks shown in Figure 2. During *training*, each application is profiled using a two-phase procedure applied to each type of system call separately. *Firstly*, a single-linkage, bottom-up, agglomerative, hierarchical clustering algorithm [6] is used to find sub-clusters of invocations with similar arguments. Anomaly models are created upon these clusters, and not on the specific system call, in order to better capture normality and deviations on a more compact input space. This is important because some system calls, most notably `open`, are used in very different ways. By exploiting effective distance models between arguments of the same type, the agglomerate system call is divided into sub-groups that are specific to a single function. For instance, invocations of `open` in `httpd` differs from those in, say, `login`. Afterwards, the system builds anomaly models of the parameters inside each cluster. It is important to note that the models used for computing distance (for clustering) and those used to build the "representation" of the cluster for anomaly detection are not necessarily the same. More details on how the distance are defined, and on the anomaly models used by S^{2}A^{2}DE, can be found in [3].

The *second phase* of training takes into account the execution *context* of each call to build a behavioral profile of programs flow. Markov chains are constructed on top of the various clusters output from the first phase: *one cluster* corresponds

to *one state* of the chain. For instance, with three clusters for the `open` syscall, and two of the `execve` syscall, then the chain is constituted by five states: $\texttt{open}_1$, $\texttt{open}_2$, $\texttt{open}_3$, $\texttt{execve}_1$, $\texttt{execve}_2$. Each transition reflects the probability of passing from one of these groups to another through the program. This approach was investigated in former literature [7,8,1,9,10], but never in conjunction with the handling of parameters and with a clustering approach.

During *training*, each execution of the program in the training set is considered as a sequence of observations. Using the output of the clustering process, each syscall is classified into the correct cluster, by computing the probability value for each model and choosing the cluster whose models give out the maximum composite probability along all known models: $\max(\prod_{i \in \mathrm{M}} P_i)$. The other probabilities are then straightforward to compute. $\mathrm{S^2A^2DE}$ is resistant to the presence of a limited number of outliers (e.g. abruptly terminated executions or attacks) in the training set, because the resulting transition probabilities will drop near zero. For the same reason, it is also resistant to the presence of any cluster of anomalous invocations created by the clustering phase. Therefore, the presence of a minority of attacks in the training set will not adversely affect the learning phase, which in turn does not require an attack-free training set, and thus it can be performed on the deployment machine.

At *detection* time, the cluster models are once again used to classify each syscall into the correct cluster. The probability value for each model is computed and the stored cluster whose models give out the maximum composite probability $P_c = \max(\prod_{i \in \mathrm{M}} P_i)$ is chosen as the correct "system call class". Anomaly thresholds are built upon two probabilities, the *punctual* probability P_p and the *sequence* probability P_s. The former is $P_p = P_c \cdot P_m$, where P_c is the probability of the *system call* to belong to the best-matching cluster and P_m is the *latest transition* probability in the chain. P_s is the probability of the whole *execution sequence* to fit the whole chain. To avoid P_s to quickly reach zero for long sequences of system calls, the probability is scaled as $P_s(l) = \sqrt[2l]{\prod_{i=1}^{l} P_p(i)^i}$, where l is the sequence length).

For both the probabilities, threshold values are equal to the lowest probability over all the training dataset, for each single application, scaled through a user-defined *sensitivity* which allows to trade off between detection rate and false positive rate. A process is flagged as malicious if either P_s or $P_p = P_c \cdot P_m$ are lower than the corresponding thresholds.

3 Enhanced Detection Models

The improvements we made focus on *path* and *execution* arguments. *String length* is now modeled using a Gaussian interval as detailed in Section 3.1. The new *edge frequency* model described in Section 3.2 have been added to detect *Denial of Service* (DoS) attacks. Also, in Section 3.3 we describe how we exploited *Self Organizing Maps* (SOMs) to model the similarity among *path arguments*. The resulting system, Hybrid IDS. incorporates the models of FSA-DF and $\mathrm{S^2A^2DE}$ along with the aforementioned enhancements.

3.1 Arguments Length Using Gaussian Intervals

The model for system call execution arguments implemented in S^2A^2DE takes into account the minimum and maximum length of the parameters found during training, and checks whether each string parameter falls into this range (model probability 1) or not (model probability 0). This technique allows to detect common attempts of buffer overflow through the command line, for instance, as well as various other command line exploits. However, such criteria do not model "how different" two arguments are to each others; a smoother function is more desirable. Furthermore, the frequency of each argument in the training set is not taken into account at all. Last but not least, the model is not resilient to the presence of attacks in the training set; just one occurrence of a malicious string would increase the length of the maximum interval allowing argument of almost every length.

The improved version of the interval model uses a Gaussian distribution for modeling the argument length $X_{args} = |args|$, estimated from the data in terms of sample mean and sample variance. The anomaly threshold is a percentile T_{args} centered on the mean. Arguments which length is *outside* the stochastic interval are flagged as anomalous. This model is resilient to the presence of outliers in the dataset. The Gaussian distribution has been chosen since is the natural stochastic extension of a range interval for the length. An example is shown in Figure 3.

Model Validation. During detection the model self-assesses its precision by calculating the kurtosis measure [11], defined as $\gamma_X = \frac{\mathrm{E}^4(X)}{\mathrm{Var}(X)^2}$. Thin tailed distributions with a low peak around the mean exhibit $\gamma_X < 0$ while positive values are typical of fat tailed distributions with an acute peak. We used $\hat{\gamma}_X = \frac{\mu_{X,4}}{\sigma_X^4} - 3$ to estimate γ_X. Thus, if $\gamma_{X_{args}} < 0$ means that the sample is spread on a big interval, while positive the values indicates a less "fuzzy" set of values. It is indeed straightforward that highly negative values indicates not significant estimations as the interval would include almost all lengths. In this case, the model falls back to a simple interval.

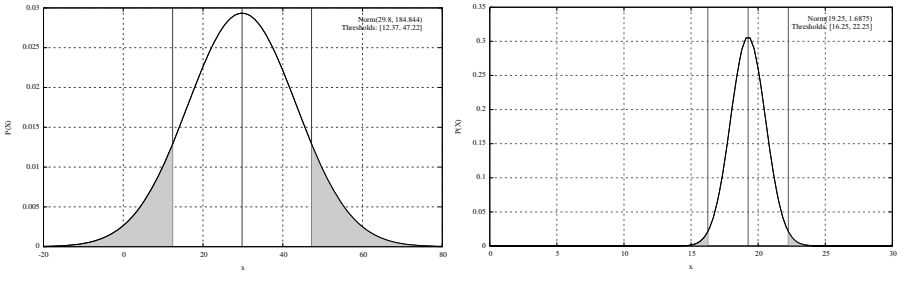

Fig. 3. Estimated Gaussian intervals for string length. Training data of `sudo` (left) and `ftp` (right) was used. $\mathcal{N}(29.8, 184.844)$, thresholds [12.37, 47.22] (left) and $\mathcal{N}(19.25, 1.6875)$, thresholds [16.25, 22.25] (right).

3.2 DoS Detection Using Edge Traversal Frequency

DoS attacks which force the process to get stuck in a legal section of the normal control flow could be detected by S^2A^2DE as violations of the Markov model, but not by FSA-DF. On the other hand, the statistical models implemented in S^2A^2DE are more robust but have higher *False Negative Rates* (FNR) than the deterministic detection implemented in FSA-DF. However, as already stated in Section 2.2, the cumulative probability of the traversed edges works well only with execution traces of similar and fixed length, otherwise even the rescaled score decreases to zero, generating false positives on long traces.

To solve these issues a stochastic model of the edge frequency traversal is used. For each trace of the training set, our algorithm counts the number of edge traversals (i.e., Markov model edge or FSA edge). The number is then normalized w.r.t. all the edges obtaining frequencies. Each edge is then associated to the sample $X_{edge} = x_1, x_2, \ldots$. We show that the random samples X_{edge} is well estimated using a Beta distribution. Figure 4 shows sample plots of this model estimated using the `mt-daapd` training set; quantiles associated to the thresholds are computed and shown as well. As we did for the Gaussian model (Section 3.1), the detection thresholds are defined at configuration time as a percentile T_{edge} centered on the mean (Figure 4). We chose the Beta for its high flexibility; a Gaussian is unsuitable to model skewed phenomena.

Model Validation. Our implementation is optimized to avoid overfitting and meaningless estimations. A model is valid only if the training set includes a significant ($|\min_i\{x_i\} - \max_i\{x_i\}| \geq \delta x_{min} = 0.04$) amount ($N^{min}_{edge} = 6$) of paths. Otherwise it construct a simpler frequency range model. The model exhibits the side effect of discarding the extreme values found in training and leads to erroneous decisions. More precisely, if the sample is $X_{edge} = 1, 1, \ldots, 0.9, 1$, the right boundary will never be exactly 1, and therefore legal values will be discarded. To solve this issue, the quantiles close to 1 are approximated to 1 according to a configuration parameter $\bar{X}_{cut}$. For instance, if $\bar{X}_{cut} = 3$ the quantile $F_X(\cdot) = 0.99\bar{9}$ is approximated to 1.

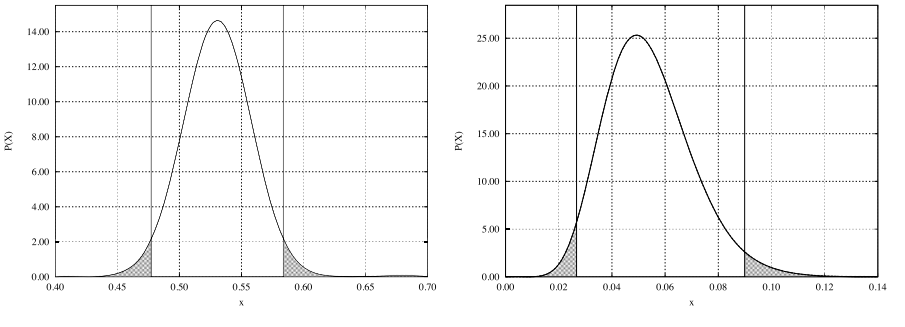

Fig. 4. Two different estimations of the edge frequency distribution. Namely, Beta(178.445, 157.866) with thresholds [0.477199, 0.583649] (left) and Beta(10.3529,181.647) with thresholds [0.0266882, 0.0899057] (right).

3.3 Path Similarity Using Self Organizing Maps

Path argument models are already implemented in S^2A^2DE and FSA-DF. Several, general-purpose string comparison techniques have been proposed so far, especially in the field of database systems and data cleansing [12]. We propose a solution based on Symbol-SOMs [13] to define an accurate distance metric between paths. Symbol SOM implements a smooth similarity measure otherwise unachievable using common, crisp distance functions among strings (e.g., edit distance).

The technique exploits *Self Organizing Maps* (SOMs), which are unsupervised neural algorithms. A SOM produces a compressed, multidimensional representation (usually a bi-dimensional *map*) of the input space by preserving the main topological properties. It is initialized randomly, and then adapted via a competitive and cooperative learning process. At each cycle, a new input is compared to the known models, and the *Best Matching Unit* (BMU) node is selected. The BMU and its neighborhood models are then updated to make them better resemble future inputs.

We use the technique described in [14] to map *strings* onto SOMs. Formally, let $S_t = [s_t(1) \cdots s_t(L)]$ denote the t-th string over the alphabet $\mathcal{A}$ of size $|\mathcal{A}|$. Each symbol $s_t(i), i = 1 \ldots L$, is then encoded into a vector $\underline{s}_t(i)$ of size $|\mathcal{A}|$ initialized with zeroes except at the w-th position which corresponds to the index of the encoded symbol (e.g., $s_t(i) =$ 'b' would be $\underline{s}_t(i) = [0\ 1\ 0\ 0 \cdots 0]^T$, $w = 2$). Thus, each string S_t is represented with sequence of L vectors like $\underline{s}_t(i)$, i.e. a $L \times |\mathcal{A}|$-matrix: $\underline{\underline{S}}_t$.

Let $\underline{\underline{S}}_t$ and $\underline{\underline{M}}_k$ denote two vector-encoded strings, where $\underline{\underline{M}}_k$ is the model associated with SOM node k. The distance between the two strings is $D'(S_t, M_k) = D(\underline{\underline{S}}_t, \underline{\underline{M}}_k)$. $D(\cdot, \cdot)$ is also defined in the case of $L_{S_t} = |S_t| \neq |M_k| = L_{M_k}$ relying on dynamic time warping techniques to find the best alignment between the two sequences before computing the distance. Without going into details, the algorithm [13] aligns the two sequences $\underline{s}_t(i) \in \underline{\underline{S}}_t, \underline{m}_k(j) \in \underline{\underline{M}}_k$ using a mapping $[\underline{s}_t(i), \underline{m}_k(j)] \mapsto [\underline{s}_t(i(p)), \underline{m}_k(j(p))]$ defined through the warping function $F : [i, j] \mapsto [i(p), j(p)]$: $F = [[i(1), j(1)], \ldots, [i(p), j(p)], \ldots, [i(P), j(P)]]$. The distance function D is defined over the warping alignment of size P, $D(\underline{\underline{S}}_t, \underline{\underline{M}}_k) = \sum_{p=1}^{P} d(i, j)$, which is $P = L_{S_t} = L_{M_k}$ if the two strings have equal lengths. $d(i, j) = d(i(p), j(p))||\underline{s}_t(i(p)) - \underline{m}_k(j(p))||$.

The distance is defined upon $g_{i,j} = g(i, j)$, the variable which stores the cumulative distance in each trellis point $(i, j) = (i(p), i(p))$. The trellis is first initialized to 0 in $(0, 0)$, to $+\infty$ for both $(0, \cdot)$ and $(\cdot, 0)$, otherwise:

$$g(i, j) = \min \begin{cases} g(i, j-1) + d(i, j) \\ g(i-1, j-1) + d(i, j) \\ g(i-1, j) + d(i, j) \end{cases}$$

Note that $i \in [1, L_{S_t}]$ and $j \in [1, L_{M_k}]$ thus the total distance is $D(\underline{\underline{S}}_t, \underline{\underline{M}}_k) = g(L_{S_t}, L_{M_k})$. A simple example of distance computation is show in Figure 5 ($\mathcal{A}$ is the English alphabet plus extra characters). The overall distance is $D'(S_t, M_k) = 8.485$. We used a symmetric Gaussian neighborhood function h whose center is

$$D(\underline{\underline{S}}_t, \underline{\underline{M}}_k) = \begin{array}{c} \\ / \\ \mathsf{v} \\ \mathsf{a} \\ \mathsf{r} \\ / \\ \mathsf{l} \\ \mathsf{o} \\ \mathsf{g} \end{array} \begin{array}{c} \begin{array}{ccccccccc} & / & \mathsf{b} & \mathsf{i} & \mathsf{n} & / & \mathsf{s} & \mathsf{h} \end{array} \\ \begin{pmatrix} 0 & +\infty & +\infty & +\infty & +\infty & +\infty & +\infty & +\infty \\ +\infty & 0 & 1.414 & 2.828 & 4.242 & 4.242 & 5.656 & 7.071 \\ +\infty & 1.414 & 1.414 & 2.828 & 4.242 & 5.656 & 5.656 & 7.071 \\ +\infty & 2.828 & 2.828 & 2.828 & 4.242 & 5.656 & 7.071 & 7.071 \\ +\infty & 4.242 & 5.656 & 5.656 & 5.656 & 4.242 & 5.656 & 7.071 \\ +\infty & 4.242 & 4.242 & 4.242 & 4.242 & 5.656 & 7.071 & 8.485 \\ +\infty & 5.656 & 5.656 & 7.071 & 7.071 & 5.656 & 5.656 & 7.071 \\ +\infty & 7.071 & 7.071 & 7.071 & 8.485 & 7.071 & 7.071 & 7.071 \\ +\infty & 8.485 & 8.485 & 8.485 & 8.485 & 8.485 & 8.485 & \mathbf{8.485} \end{pmatrix} \end{array}$$

Fig. 5. Distance computation example between `/bin/sh` an `/var/log`

located at the BMU $c(t)$. More precisely, $h(k, c(t), t) = \alpha(t)e^{-\frac{d(c(t),k)}{2\sigma^2(t)}}$, where $\alpha(t)$ controls the learning rate and $\sigma(t)$ is the actual width of the neighborhood function. The SOM algorithm uses *two* training cycles. During (1) *adaptation* the map is more flexible, while during (2) *tuning* the learning rate $\alpha_{(\cdot)}$ and the width of the neighborhood $\sigma_{(\cdot)}$ are decreased. On each phase such parameters are denoted as $\alpha_1, \alpha_2, \sigma_1, \sigma_2$.

Symbol SOMs are "plugged" into FSA-DF by associating each transition with the *set* of BMUs learned during training. At detection, an alert occurs whenever a path argument falls into neighborhood of a non-existing BMU. Similarly, in the case of S^2A^2DE, the neighborhood function is used to decide whether the string is anomalous or not, according to a proper threshold which is the minimum value of the neighborhood function encountered during training, for each node.

4 Experimental Evaluation

In this section we describe our efforts to cope with the lack of reliable testing datasets for intrusion detections. The testing methodology is here detailed along with the experiments we designed. Both detection accuracy and performance overhead are subjects of our tests.

4.1 Testing Methodology and Data Generation

Comparing and benchmarking IDSes is a well known problem [15]. Since the commonly used DARPA evaluation datasets exhibit well known shortcomings, we decided to generate a new dataset. We chose a number of recent exploits from CVE, including different types of vulnerabilities (code injections, file writes, denial of service attacks) as well as attacks that easily evade existing IDSes by slightly modifying the data flows and not the control flows. Clean training data was obtained by collecting benign system calls sequences during the normal execution of the target applications. We used attacks against `sing`, `mt-daapd`, `proftpd`, `sudo`, and `BitchX`. We refer to the vulnerabilities by their *Common Vulnerabilities Exposures* (CVE) ID.

Table 1. Parameters used to train the IDSes. Values includes the number of traces used, the amount of paths encountered and the number of paths per cycle.

	`sing`	`mt-daapd`	`proftpd`	`sudo`	`BitchX`	`mcweject`	`bsdtar`
SOM size	15×15	15×15	15×15	15×15	10×10	15×15	15×15
Traces	18	18	18	18	14	10	240
Syscalls	5808	194879	64640	52034	103148	84	12983
Paths	2700	2700	23632	1316	14921	48	3477
Paths/cycle%	2	2	1	8	1	50	2

Specifically, `sing` is affected by CVE-2007-6211, a vulnerability which allows to write arbitrary text on arbitrary files by exploiting a combination of parameters. This attack is meaningful because it does not alter the control flow, but just the data flow, with an `open` which writes on unusual files. Training datasets contain traces of regular usage of the program invoked with large sets of command line options.

`mt-daapd` is affected by a format string vulnerability (CVE-2007-5825) in `ws_addarg()`. It allows remote execution of arbitrary code by including the format specifiers in the username or password portion of the base64-encoded data on the `Authorization: Basic` HTTP header sent to `/xml-rpc`. The `mod_ctrls` module of `proftpd` let local attackers to fully control the integer `regarglen` (CVE-2006-6563) and exploit a stack overflow to gain root privileges.

`sudo` does not properly sanitize data supplied through `SHELLOPTS` and `PS4` environment variables, which are passed on to the invoked program (CVE-2005-2959). This leads to the execution of arbitrary commands as privileged user, and it can be exploited by users who have been granted limited superuser privileges. The training set includes a number of execution of programs commonly run through sudo (e.g., `passwd`, `adduser`, editing of `/etc/` files) by various users with different, limited superuser privileges, along with benign traces similar to the attacks, invoked using several permutations of option flags.

`BitchX` is affected by CVE-2007-3360, which allows a remote attacker to execute arbitrary commands by overfilling a hash table and injecting an EXEC hook function which receives and executes shell commands. Moreover, failed exploit attempts can cause DoS. The training set includes several IRC client sessions and a legal IRC session to a server having the same address of the malicious one.

In order to evaluate and highlight the impact of each specific model, we performed targeted tests rather than reporting general DRs and FPRs only. Also, we ensured that all possible alerts types are inspected (i.e., true/false positive/negative). In particular, for each IDS, we included one *legal* trace in which file operations are performed on files *never* seen during training but with a similar name (e.g., training on `/tmp/log`, testing on `/tmp/log2`); secondly, we inserted a trace which mimics an attack.

4.2 Comparison of Detection Accuracy

The detection accuracy of Hybrid IDS (H), FSA-DF (F) and S^{2}A^{2}DE (S) is here analyzed and compared. Both training parameters and detection results are summarized in Table 1. The parameters used to train the SOM are fixed except for $\sigma_1(t)$: $\alpha_1(t) = 0.5 \div 0.01$, $\sigma_2(t) = 3$ and $\alpha_2(t) = 0.1 \div 0.01$. Percentiles for both X_{args} and X_{edge} are detailed. The "paths/cycle%" (paths per cycle) row indicates the amount of paths arguments used for training the SOM. The settings for clustering stage of S^{2}A^{2}DE are constant: minimum number of clusters (3, or 2 in the case of the `open`); maximum merging distance (6, or 10 in the case of the `open`); the "null" and the "don't care" probability values are fixed at 0.1 and 10, respectively, while 10 is the maximum number of leaf clusters. In order to give a better understanding of how each prototype works, we analyzed by hand the detection results on each target application.

`sing`: Hybrid IDS is not tricked by the false positive mimic trace inserted. The Symbol SOM model recognizes the similarity of `/tmp/log3` with the other paths inserted in the training. Instead, both FSA-DF and S^{2}A^{2}DE raise false alarms; the former has never seen the path during training while the latter recognizes the string in the tree path model but an alarm is raised because of threshold violation. S^{2}A^{2}DE recognizes the attack containing the longer subsequent invocations of `mmap2`; FSA-DF also raises a violation in the file name because it has never been trained against `/etc/passwd` nor `/etc/shadow`; and Hybrid IDS is triggered because the paths are placed in a different SOM region w.r.t. the training.

`mt-daapd`: The legit traces violate the binary and unary relations causing several false alarms on FSA-DF. On the other hand, the smoother path similarity model allows Hybrid IDS and S^{2}A^{2}DE to pass the test with no false positives. The changes in the control flow caused by the attacks are recognized by all the IDSes. In particular, the DoS attack (special-crafted request sent fifty times) triggers an anomaly in the edge frequency model.

`proftpd`: The legit trace is correctly handled by all the IDSes as well as the anomalous root shell that causes unexpected calls (`setuid`, `setgid` and `execve`) to be invoked. Howerver, FSA-DF flags more than 1000 benign system calls as anomalous because of temporary files path not present in the training.

`sudo`: Legit traces are correctly recognized by all the engines and attacks are detected without errors. S^{2}A^{2}DE fires an alert because of a missing edge in the Markov model (i.e., the unexpected execution of `chown root:root script` and `chmod +s script`). Also, the absence of the `script` string in the training triggers a unary relation violation in FSA-DF and a SOM violation in Hybrid IDS. The traces which mimic the attack are erroneously flagged as anomalous, because the system call sequences are *strictly* similar to the attack.

`BitchX`: The exploit is easily detected by all the IDSes as a control flow violation through extra `execve` system calls are invoked to execute injected commands. Furthermore, the Hybrid IDS anomaly engine is triggered by three edge frequency violations due to paths passed to the FSA when the attack is performed which are different w.r.t. the expected ones.

4.3 Specific Comparison of SOM-S^{2}A^{2}DE and S^{2}A^{2}DE

We also specifically tested how the introduction of a Symbol SOM improves over the original probabilistic tree used for modeling the path arguments in S^{2}A^{2}DE. As summarized in right side of Table 2, the FPR decreases in the second test. However, the first test exhibits a lower FNR as detailed in the following.

The `mcweject` utility is affected by a stack overflow CVE-2007-1719 caused by improper bounds checking. Root privileges can be gained if `mcweject` is `setuid`. The exploit is as easy as `eject -t illegal_payload`, but we performed it through userland exec [16] to make it more silent avoiding the `execve` that obviously triggers an alert in the S^{2}A^{2}DE for a missing edge in the Markov chain. Instead, we are interested in comparing the string models only. SOM-S^{2}A^{2}DE detects it with no issues because of the use of different "types" of paths in the `open`s.

An erroneous computation of a buffer length is exploited to execute code via a specially crafted PAX archives passed to `bsdtar` (CVE-2007-3641). A heap overflow allows to overwrite a structure pointer containing itself another pointer to a function called right after the overflow. The custom exploit [16] basically redirects that pointer to the injected shellcode. Both the original string model and the Symbol SOM models detect the attack when the unexpected special file `/dev/tty` is opened. However, the original model raises many false positives when significantly different paths are encountered. This situation is instead handled with no false positives by the smooth Symbol SOM model.

4.4 Performance Evaluation and Complexity Discussion

We performed both empirical measurements and theoretical analysis of the performance of the various proposed prototypes. Detection speed results are summarized in Table 3. The datasets for detection accuracy were reused: we selected

Table 2. Comparison of the FPR of S^{2}A^{2}DE vs. FSA-DF vs. Hybrid IDS and S^{2}A^{2}DE vs. SOM-S^{2}A^{2}DE. Values include the number of traces used. Accurate description of the impact of each *individual* model is in Section 4.2 (first five columns) and 4.3 (last two columns).

	`sing`	`mt-daapd`	`profdtpd`	`sudo`	`BitchX`	`mcweject`	`bsdtar`	
Traces	22	18	21	22	15	12	2	
Syscalls	1528	9832	18114	3157	107784	75	102	
S^{2}A^{2}DE	10.0%	0%	0%	10.0%	0.0%	0.0%	8.7%	**S^{2}A^{2}DE**
FSA-DS	5.0%	16.7%	28%	15.0%	0.0%			
Hybrid IDS	0.0%	0%	0%	10.0%	0.0%	0.0%	0.0%	**SOM-S^{2}A^{2}DE**

Table 3. Detection performance measured in μsec/syscall. The average speed is measured in syscall/sec (last column).

	`sing`	`sudo`	`BitchX`	`mcweject`	`bsdtar`	**Avg. speed**
System calls	3470	15308	12319	97	705	
S^{2}A^{2}DE	115.3	52.26	154.2	1030	141.8	8463
FSA-DF	374.6	97.98	97.41	-	-	7713
Hybrid IDS	7492	378.8	2167	-	-	1067
SOM-S^{2}A^{2}DE	-	-	-	90721	26950	25

the five test applications on which the IDSes performed worst. Hybrid IDS is slow because the BMU algorithm for the symbol SOM is invoked for each system call with a path argument (`open`s are quite frequent), slowing down the detection phase. Also, we recall that the current prototype relies on a system call interceptor based on `ptrace` which *introduces high runtime overheads*, as shown in [2]. To obtain better performance, an in-kernel interceptor could be used. The theoretical performance of each engine can be estimated by analyzing the bottleneck algorithm.

Complexity of FSA-DF. During training, the bottleneck is the binary relation learning algorithm. $T_F^{train} = O(S \cdot M + N)$, where M is the total number of system calls, $S = |Q|$ is the number of states of the automaton, and N is the sum of the length of all the string arguments in the training set. At detection $T_{FSA-DF}^{det} = O(M + N)$.

Assuming that each system call has $O(1)$ arguments, the training algorithm is invoked $O(M)$ times. The time complexity of each i-th iteration is $Y_i + |X_i|$, where Y_i is the time required to compute all the unary and binary relations and $|X_i|$ indicates the time required to process the $i - th$ system call X. Thus, the overall complexity is bounded by $\sum_{i=1}^{M} Y + |X_i| = M \cdot Y + \sum_{i=1}^{M} |X_i|$. The second factor $\sum_{i=1}^{M} |X_i|$ can be simplified to N because strings are represented as a tree; it can be shown [2] that the total time required to keep the longest common prefix information is bounded by the total length of all input strings. Furthermore, Y is bounded by the number of unique arguments, which in turn is bounded by S; thus, $T_F^{train} = O(S \cdot M + N)$. This also prove the time complexity of the detection algorithm which, at each state and for each input, requires unary and binary checks to be performed; thus, its cost is bounded by $M + N$. ■

Complexity of Hybrid IDS. In the training phase, the bottleneck is the Symbol SOM creation time: $T_H^{train} = O(C \cdot D \cdot (L^2 + L))$, where C is the number of learning cycles, D is the number of nodes, and L is the maximum length of an input string. At detection time $T_H^{det} = O(M \cdot D \cdot L^2)$.

T_H^{train} depends on both the number of training cycles, the BMU algorithm and node updating. The input is randomized at each training session and a constant amount of paths is used, thus the input size is $O(1)$. The BMU algorithm depends on both the SOM size and the distance computation,

bounded by $L_{input} \cdot L_{node} = L^2$, where L_{input} and L_{node} are the *lengths* of the input string and the node string, respectively. More precisely, the distance between strings is performed by comparing all the vectors representing, respectively, each character of the input string and each character of the node string. The char-by-char comparison is performed in $O(1)$ because the size of each character vector is fixed. Thus, the distance computation is bounded by $L^2 \simeq L_{input} \cdot L_{node}$. The node updating algorithm depends on both the number of nodes D, the length of the node string L_{node} and the training cycles C, hence each cycle requires $O(D \cdot (L^2 + L))$, where L is the length of the longest string. The creation of the FSA is similar to the FSA-DF training, except for the computation of the relations between strings which time is no longer $O(N)$ but it is bounded by $M \cdot D \cdot L^2$ (i.e., the time required to find the Best Matching Unit for one string). Thus, according to *Proof 1*, this phase requires $O(S \cdot M + M \cdot D \cdot L^2) < O(C \cdot D \cdot (L^2 + L))$. The detection time T_H^{det} is bounded by the BMU algorithm, that is $O(M \cdot D \cdot L^2)$. ■

The clustering phase of **S²A²DE** is $O(N^2)$ while with **SOM-S²A²DE** it grows to $O(N^2 L^2)$.

In the worst case, the clustering algorithm used in [3] is known to be $O(N^2)$, where N is the number of system calls: the distance function is $O(1)$ and the distance matrix is searched for the two closest clusters. In the case of SOM-S²A²DE, the distance function is instead $O(L^2)$ as it requires one run of the BMU algorithm. ■

5 Related Work

Due to space limitations we focus on the subset of literature which uses unsupervised learning algorithms for anomaly detection over system calls. We refer the reader to [17] for a more comprehensive and taxonomic review.

The first mention of intrusion detection through the analysis of the sequence of syscalls from system processes is in [18], where "normal sequences" of system calls (similar to N-grams) are considered (ignoring the parameters of each invocation). Variants of [18] have been proposed in [19,7,1]; this type of techniques can also be used a reactive, IPS-like fashion [20]. The core assumption is that intrusions generate sequences of system calls that are unusual during normal application usage. Each sequence of system calls is tokenized in substrings using a sliding window of N elements. All the substrings seen in training are stored; during detection, any N-gram never seen before raises an alarm. The precision of this method depends on the value chosen for N. A low value of N tends to generate false negatives (the worst-case scenario, $N = 1$, only checks if a system call was already seen during training).

FSA have also been used to express the language of the system calls of a program, using either deterministic [21] or non-deterministic [22] automata. The issue when using FSA is how to define the states of the machine: at the highest

level of detail, each state is linked to a specific instruction of the program, while transitions are usually identified with system calls. An FSA improves over the N-gram model with better efficiency and, in addition, it does not suffer from the choice of arbitrary parameter N.

A static analysis approach to extract a call graph was proposed in [23]. Giffin et al. [24] developed a different version of this approach, based on the analysis of the binaries, integrating the execution environment as a model constraint. However, static analysis approaches such as these follow all possible execution paths, therefore they are conservative and may include additional, extraneous control flows; they may also leave more way for mimicry attacks. On the other hand, automatically generating a compact FSA representation from system call traces is not an easy task. A similar method [25] uses pushdown automata to enrich the model with a "stack" structure, which is used to choose each next transition to take, and can be manipulated as part of the transition. In Section 2.1 we described more in depth an IDS based on this approach [2] which uses the program counter to define states and syscalls as transitions, but complements them with *dataflow* information. However, all these methods suffer from an inherent brittleness: if the training is insufficient, a number of false positives could be generated because the models are extremely narrow. The use of *Hidden Markov Models* (HMMs) has also been proposed to model sequences of system calls [9]. In [26] HMMs are compared with the models used in [19,20] and shown to perform considerably better, even if with an added computational overhead; unfortunately, the datasets used for the comparative evaluation are no longer available for comparison. Using Markov chains instead of hidden models decreases this overhead, as observed in [27]. In [28] HMMs are observed to perform considerably better than FSA and similar models. The main difference of these models stochastic part: the transitions are not deterministic but linked to a probability and this could allow a reduction of the FPR. In Section 2.2 we analyzed S^2A^2DE [3], a HIDS based on this approach, but which complements it with anomaly models built on syscall arguments.

The two systems analyzed in Section 2 also take into account the parameters of the system calls. Even if this is an inherently complex task, it has been already proven to yield a lot of potential. For instance, mimicry attacks [29] can evade the detection of syscall sequence anomalies, but it is much harder to devise ways to cheat both the analysis of sequence and arguments. Besides the ones we discuss in the following, two other recent research works focused on this problem. Another example is [30] in which a number of models are introduced to deal with the most common arguments, even if without caring for the sequence of system calls. In [31] the LERAD algorithm (Learning Rules for Anomaly Detection) is used to mine rules expressing "normal" values of arguments, normal sequences of system calls, or both. However, no relationship among the values of different arguments is learned; sequences and argument values are handled separately; the evaluation is quite poor however, and uses non-standard metrics.

6 Conclusions

We have presented two alternative, state-of-the-art approaches for anomaly detection over system call sequences and arguments: a deterministic IDS which builds an FSA model complemented by a network of dataflow relationships among the system call arguments (which we nicknamed FSA-DF), and a prototype named S^2A^2DE which builds a Markov chain of the system calls, complementing it with several models for detecting anomalies in the parameters and clustering system calls according to their content. We showed how the model for system call execution arguments implemented in S^2A^2DE can be improved by using better statistical models. We also proposed a new model for counting the frequency of traversal of edges on the FSA prototype, to make it able to detect denial-of-service attacks. Both systems needed an improved model for string (path) similarity. We adapted the Symbol SOM algorithm to make it suitable for computing a "distance" between two paths. We believe that this is the core contribution of the work.

We tested and compared the original prototypes with an hybrid solution where the Symbol SOM and the edge traversal models are applied to the FSA, and a version of S^2A^2DE enhanced with the Symbol SOM and the correction to the execution arguments model. Both the new prototypes have the *same* detection rates of the original ones, but significantly *lower* false positive rates. This is paid in terms of a non-negligible limit to detection speed, at least in our proof of concept implementation.

Future extensions of this work will re-engineer the prototypes to use an in-kernel system call interceptor, and generically improve their performance. We are studying how to speed up the Symbol SOM node search algorithm, in order to bring the throughput to a rate suitable for online use.

References

1. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security 6(3), 151–180 (1998)
2. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. In: IEEE Symposium on Security and Privacy, May 2006, pp. 15–62 (May 2006)
3. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Transactions on Dependable and Secure Computing (accepted for publication)
4. Sharif, M.I., Singh, K., Giffin, J.T., Lee, W.: Understanding precision in host based intrusion detection. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 21–41. Springer, Heidelberg (2007)
5. Zanero, S.: Unsupervised Learning Algorithms for Intrusion Detection. PhD thesis, Politecnico di Milano T.U., Milano, Italy (May 2006)
6. Han, J., Kamber, M.: Data Mining: concepts and techniques. Morgan-Kauffman, San Francisco (2000)
7. Cabrera, J.B.D., Lewis, L., Mehara, R.: Detection and classification of intrusion and faults using sequences of system calls. ACM SIGMOD Record 30(4) (2001)

8. Casas-Garriga, G., Díaz, P., Balcázar, J.: ISSA: An integrated system for sequence analysis. Technical Report DELIS-TR-0103, Universitat Paderborn (2005)
9. Ourston, D., Matzner, S., Stump, W., Hopkins, B.: Applications of hidden markov models to detecting multi-stage network attacks. In: HICSS, p. 334 (2003)
10. Jha, S., Tan, K., Maxion, R.A.: Markov chains, classifiers, and intrusion detection. In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW 2001), Washington, DC, USA, June 2001, pp. 206–219. IEEE Computer Society Press, Los Alamitos (2001)
11. Joanes, D., Gill, C.: Comparing Measures of Sample Skewness and Kurtosis. The Statistician 47(1), 183–189 (1998)
12. Elmagarmid, A., Ipeirotis, P., Verykios, V.: Duplicate Record Detection: A Survey. IEEE Transactions on Knowledge and Data Engineering 19(1), 1–16 (2007)
13. Somervuo, P.J.: Online algorithm for the self-organizing map of symbol strings. Neural Netw. 17(8-9), 1231–1239 (2004)
14. Kohonen, T., Somervuo, P.: Self-organizing maps of symbol strings. Neurocomputing 21(1-3), 19–30 (1998)
15. Zanero, S.: Flaws and frauds in the evaluation of IDS/IPS technologies. In: Proc. of FIRST 2007 - Forum of Incident Response and Security Teams, Sevilla, Spain (June 2007)
16. Maggi, F., Zanero, S., Iozzo, V.: Seeing the invisible - forensic uses of anomaly detection and machine learning. ACM Operating Systems Review (April 2008)
17. Bace, R.G.: Intrusion detection. Macmillan Publishing Co., Inc., Indianapolis (2000)
18. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for Unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, Washington, DC, USA. IEEE Computer Society, Los Alamitos (1996)
19. Forrest, S., Perelson, A.S., Allen, L., Cherukuri, R.: Self-nonself discrimination in a computer. In: SP 1994: Proceedings of the 1994 IEEE Symposium on Security and Privacy, Washington, DC, USA, p. 202. IEEE Computer Society, Los Alamitos (1994)
20. Somayaji, A., Forrest, S.: Automated response using system–call delays. In: Proceedings of the 9th USENIX Security Symposium, Denver, CO (August 2000)
21. Michael, C.C., Ghosh, A.: Simple, state-based approaches to program-based anomaly detection. ACM Trans. Inf. Syst. Secur. 5(3), 203–237 (2002)
22. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Washington, DC, USA. IEEE Computer Society Press, Los Alamitos (2001)
23. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: SP 2001: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Washington, DC, USA, pp. 156–168. IEEE Computer Society Press, Los Alamitos (2001)
24. Giffin, J.T., Dagon, D., Jha, S., Lee, W., Miller, B.P.: Environment-sensitive intrusion detection. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 185–206. Springer, Heidelberg (2006)
25. Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proceedings. 2003 Symposium on Security and Privacy, 2003, May 11-14, pp. 62–75 (2003)
26. Warrender, C., Forrest, S., Pearlmutter, B.A.: Detecting intrusions using system calls: Alternative data models. In: IEEE Symposium on Security and Privacy, pp. 133–145 (1999)

27. Jha, S., Tan, K., Maxion, R.A.: Markov chains, classifiers, and intrusion detection. In: CSFW 2001: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, pp. 206–219. IEEE Computer Society, Washington (2001)
28. Yeung, D.Y., Ding, Y.: Host-based intrusion detection using dynamic and static behavioral models. Pattern Recognition 36, 229–243 (2003)
29. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: CCS 2002: Proceedings of the 9th ACM conference on Computer and communications security, pp. 255–264. ACM, New York (2002)
30. Krügel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003)
31. Tandon, G., Chan, P.: Learning rules from system call arguments and sequences for anomaly detection. In: ICDM Workshop on Data Mining for Computer Security (DMSEC), pp. 20–29 (2003)