Detecting Intrusions through System Call Sequence and Argument Analysis

Federico Maggi, *Student Member, IEEE,* Matteo Matteucci, *Member, IEEE,* and Stefano Zanero, *Member, IEEE*

***Abstract*— We describe an unsupervised host-based intrusion detection system based on system calls arguments and sequences. We define a set of anomaly detection models for the individual parameters of the call. We then describe a clustering process which helps to better fit models to system call arguments, and creates inter-relations among different arguments of a system call. Finally, we add a behavioral Markov model in order to capture time correlations and abnormal behaviors. The whole system needs no prior knowledge input; it has a good signal to noise ratio, and it is also able to correctly contextualize alarms, giving the user more information to understand whether a true or false positive happened, and to detect global variations over the entire execution flow, as opposed to punctual ones over individual instances.**

***Index Terms*— Intrusion Detection; Anomaly Detection; Behavior Detection; Markov Models**

I. Introduction

THE "misuse based" approach to intrusion detection, which tries to directly define and enumerate each possible type of attack, is nowadays showing its limits. The growing number of new vulnerabilities discovered every day, and the unknown number of discovered but undisclosed vulnerabilities (the so called "zero-days") make the concept of a "knowledge base of the attacks" increasingly inefficient, and hopelessly incomplete. The polymorphism of modern attacks, and the increasing number of targeted attacks, designed to hit one particular system, further underline the insufficiency of this traditional paradigm.

The obvious solution to this problem would be a shift towards the paradigm of anomaly detection, modeling what is *normal* instead of what is *anomalous*; this is surprisingly similar to the earliest conceptions of what an IDS should do [1]. Since then, a number of host-based anomaly detection systems have been proposed in academic projects, but have been less useful than they were supposed to be in real world systems (with a few notable exceptions). This is mainly due to two undesirable properties of such systems in real world applications: firstly, their signal to noise ratio turns out to be lower than it is when they are benchmarked on well known datasets; in second place, they do not usually help the user in understanding *what* is wrong, giving just a measure of "abnormality" as output.

In this paper, we propose a novel host-based IDS which uses the sequence as well as the parameters of the system calls executed by a process to identify anomalous behaviors. The use of system calls as anomaly indicators is well established in literature (e.g. in [2]–[14]), usually without handling their parameters (with the notable exceptions of [15]–[17]). We can identify at least four key novel contributions of this paper:

- we build and carefully test anomaly detection models for system call parameters, in a similar way to [15];
- we introduce the concept of *clustering* arguments in order to automatically infer different ways to use the same system call; this leads to more precise models of normality on the arguments;
- the same concept of clustering also creates correlations among the different parameters of a same system call, which is not present in any form in [15]–[17];
- a traditional detection approach, based on deviations from previously learned Markov models of sequences, is complemented with the concept of clustering; the sequence of system calls is transformed into a sequence of labels (i.e., classes of calls): this is conceptually different than what has been done in other works (such as [16]), where sequences of events and single events by themselves are both taken into account but in an orthogonal way.

The resulting model is also able to correctly contextualize alarms, providing the user with more information to understand what caused any false positive, and to detect variations over the execution *flow*, as opposed to variations over *single* system call. We also discuss in depth how we performed the implementation and the evaluation of the prototype, trying to identify and overcome the pitfalls associated with the usage of the IDEVAL dataset.

The remainder of the paper is organized as follows: in Section II we describe previous related works; in Section III we analyze SyscallAnomaly, an earlier prototype, and describe the issue we identified in it. Section IV presents our system. In Section V we analyze the performance of our prototype. Finally, in Section VI we draw our conclusions and outline some future extensions of this work.

II. Host-based intrusion detection: state of the Art

Due to space limitations, we do not even attempt to review all of the previous literature on intrusion detection, focusing only on works dealing with host-based intrusion detection, and in particular with anomaly detection over system calls. We refer the reader to [18] for a more comprehensive and taxonomic review.

Anomaly detection has been part of intrusion detection since its very inception: it already appears in the seminal work by Anderson [1]. However, Denning was the first to actually define a set of statistical characterization techniques for events, variables and counters such as the CPU load and the usage of certain commands [19]. At that time, obviously, *host-based* techniques were the focus, and a wide literature ensued. Some works used

Manuscript received XXXXX XX, 2007; revised XXXXXX XX, 2007.

All the authors are with the Dipartimento di Elettronica e Informazione of the Politecnico di Milano T.U.

more complex, purely statistical techniques [20], [21], sometimes with very interesting results. Most of these works, however, do not take into account a sequence of events, but either just atomic events, or system-wide variables.

Other early studies focused on terminal-based access to shared servers, using the sequence of commands ran by users as a data source and trying to find out masqueraders. For instance, neural networks have been used to analyze interactive user sessions [22]. Another interesting approach [23] uses a command/user incidence matrix, which is searched for structural zeroes representing rare commands. In [24] a multi-layer perceptron is trained to recognize buffer overflows in arguments passed to a vulnerable program on the command line. A common critique that can be drawn against many published works in this particular area is the fact that nowadays users do not interactively login on remote systems as much as in the past.

Artificial immune system have been proposed as a computational approach for solving a wide range of problems [25], among which intrusion detection [18], [26]. While these approaches never became mainstream, they often dealt with system call sequences and usage as one of the key indicators of the interactions between programs and operating systems.

The first mention of intrusion detection through the analysis of the sequence of syscalls from system processes is in [2], where "normal sequences" of system calls (similar to n-grams) are considered (ignoring the parameters of each invocation). A similar idea was presented earlier in [27], but with the assumption that it would be possible to manually describe the canonical sequence of calls of each and every program, something evidently impossible in practice. However, an interesting element of this paper is that it takes into account the values of the arguments of syscalls. Variants of [2] have been proposed in [3], [4]. This type of techniques have also been proposed for reactive components [5].

An inductive rule generator called RIPPER [6] has been proposed for analyzing sequences of syscalls and extracting rules describing normal sequences of system calls [7], [28]. Alternative, supervised approaches based on data mining are presented in [29]. These approaches have the advantage of giving insights on how the features can be selected, how they interact with each other, and on the appropriate models to fit them. On the other hand, they cannot be used online to protect running systems, but just in a batch, forensics mode.

The use of *Hidden Markov Models* (HMMs) has also been proposed to model sequences of system calls [8]. In [30] HMMs are compared with the models used in [5], [28], [31] 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 [9]. In [32] we proposed a general Bayesian framework for *behavior detection* based on hints drawn from the quantitative methods of ethology and behavioral sciences.

Alternatively, other authors proposed to use static analysis, as opposed to dynamic learning, to profile a program's normal behavior. Finite state automata have been used to express the language of the system calls of a program, using deterministic [10] or non-deterministic [11] automata, or other representations, such as a call graph [12]. Giffin et al. [13] developed a different version of this approach, based on the analysis of the binaries, and integrating the execution environment as a model constraint. However, in [14] HMMs are observed to perform considerably better than static analysis models.

It is remarkable that none of these methods analyzes either the arguments or the return values of the system calls. This is due to the inherent complexity of the task, but undoubtedly the arguments contain a wide range of information that can be useful for intrusion detection. For instance, mimicry attacks [33] 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. Three recent research works began to focus on this problem. In [15] a number of models are introduced to deal with the most common arguments, without caring for the sequence of system calls. This is the work we discuss in depth and extend in our paper.

In [16] 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. No relationship is learned among the values of different arguments; sequences and argument values are handled separately; the evaluation is quite poor however, and uses non-standard metrics. A much more interesting approach is presented in [17], where a dataflow anomaly detection framework is developed, which learns rules describing the flow of information between the arguments of system calls. This approach has really interesting properties, among which the fact that not being stochastic useful properties can be demonstrated in terms of detection assurance. On the other hand, though, the set of relationships that can be learned is limited (whereas the use of unsupervised learning models such as the ones we propose in this paper can lead to the discovery of previously unknown relationships). The relations are all deterministic, which leads to a brittle detection model potentially prone to false positives. Finally, it does not discover any type of relationship between different arguments of the same call.

III. System Call Argument Analysis: the LibAnomaly framework

LibAnomaly [15], [34] is a library created to implement anomaly detection models. Using this library, a system called SyscallAnomaly has been implemented, which can detect anomalies by analyzing system call arguments. Both have been developed by the Reliable Software Group of the University of California, Santa Barbara. In the remainder of this section, we briefly describe these projects, to let the reader better understand the improvements we suggest.

A. The LibAnomaly Framework

In LibAnomaly, a generic anomaly detection model is characterized by the following properties:

- a number of elements from a *training set* can be added to the model during a training phase;
- there is an algorithm to create the model out of a given training set;
- for any given (new) input we can calculate a likelihood rating (i.e., the probability of it being generated by the model)

A *confidence rating* can also be computed at training time for any model, by determining how well it fits its training set; this value can be used at runtime to provide additional information on the reliability of the model. By using cross-validation, an *overfitting rating* can also be optionally computed. The four basic types of models implemented by LibAnomaly are the *String Length model*, the *Character Distribution model*, the *Structural Inference model* and the *Token Search model*.

The **String Length model** computes, from the strings seen in the training phase, the sample mean μ and variance σ^2 of their lengths. In the detection phase, given l, the length of the observed string, the likelihood p of the input string length with respect to the values observed in training is equal to one if $l < \mu + \sigma$ and $\frac{\sigma^2}{(l-\mu)^2}$ otherwise.

The **Character Distribution model** analyzes the discrete probability distribution of characters in a string. At training time, the so called ideal character distribution is estimated [34]: each string is considered as a set of characters, which are inserted into an histogram, in decreasing order of occurrence, with a classical rank order/frequency representation. During the training phase, a compact representation of mean and variance of the frequency for each rank is computed. For detection, a χ^2 Pearson test returns the likelihood that the observed string histogram comes from the learned model.

The **Structural Inference model** tries to learn the structure of strings. These are simplified before the analysis, using the following translation rules: `[A-Z]` $\rightarrow$ `A`, `[a-z]` $\rightarrow$ `a`, `[0-9]` $\rightarrow$ `0`. In other words, uppercase characters, lowercase characters, and numbers, are lumped together, while other characters are kept. Finally, multiple occurrences of the same character are simplified. For instance, `/usr/lib/libc.so` is translated into `/aaa/aaa/aaaa.aa`, and further compressed into `/a/a/a.a`. Strings that after this compression are still longer than 40 characters are ignored by the model, perhaps for simplification. Accepted strings are used to generate a probabilistic grammar by means of a Markov model induced by exploiting a Bayesian merging procedure, as described in [35], [36]. However, such merging is heavily dependent on the choice of a good prior for the Bayesian model, and this choice is not well documented in the literature of LibAnomaly. Curiously, the probability values associated with such Markov models (which create a sort of probabilistic grammar) are ignored in the detection phase. More precisely, the compressed string is compared with the Markov model; if the string can be generated by the model (i.e. the product of the probabilities of the traversed transitions has a value greater than 0) a probability of 1 is returned, otherwise 0 is returned. This choice is probably motivated by the fact that the different length of the observed strings would otherwise bias the probabilities returned by the model. The approach avoids the penalization of longer observations against shorter ones. We found a similar problem in our algorithm for threshold probability calculation over sequences of system calls: as detailed in Section IV-D, we addressed such issues by means of an appropriately chosen scaling function.

The **Token Search model** is applied to arguments which contain flags or modes. During training, this model uses a statistical test to determine whether or not an argument contains a finite set of values. The core idea (drawn from [37]) is that if the set is finite, then the number of different arguments in the training set will grow in a much slower way than the total number of samples. This is tested using a Kolgomorov-Smirnov non parametric test. If the field contains a set of tokens, the set of values observed during training is stored. During detection, if the field has been flagged as a token, the input is compared against the stored values list. If it matches a former input, the model returns 1, else it returns 0, without regard to the relative frequency of the tokens in the training data.

B. SyscallAnomaly

SyscallAnomaly uses LibAnomaly's models to create a profile of system calls for each different application. For each execution of an application, the input of SyscallAnomaly is a sequence of system calls, $S = [s_1, s_2, s_3, \ldots]$, logged by the operating system. Each system call s_i is characterized by a type, a list of arguments, a return value, and a timestamp.

During the training phase, SyscallAnomaly generates a profile for each possible system call type (e.g. `read`, `write`, `exec`, ...), for each application (e.g. `sendmail`, `telnetd`, ...). It does not take into account the sequence with which the system calls happen. This profile strives to capture the normal behavior of a program, by "learning" the normal arguments of each system call type inside that program, by the means of a set of models, as described above. During the detection phase, the stored models return the likelihood of a particular value of an argument for a system call, based on previous observations of that system call in the context of the same application during training.

Each model operates independently on each argument of the system call. As detailed in the following, the probabilities are then aggregated to compute the total probability value of a system call; if this value is lower than a threshold, the call is flagged as anomalous. The threshold is computed by incrementing the maximum anomaly value over the whole training set of a user-defined percentage, which is a sensitivity parameter used to tune the system.

Basically, SyscallAnomaly bases its structure on two major assumptions:

- Attacks actually appear in, and have some effect on, system calls arguments, rather than on their sequence. Attacks that do not alter the content of system calls but just their sequence are undetectable by such a system.
- Anomalous system call arguments differ from training values more than training values differ among themselves. Thus, the ability of detecting anomalies, even if the first assumption is satisfied, depends on the efficacy of at least a few of the various individual models built upon arguments to detect outliers (separately, since no correlation among models is taken into account).

As we will show in the following, the first assumption proves to be too strong. Some attacks are detectable by means of parameter content only, some by means of sequence only, and some can be detected only by combining the two methods. The second assumption is more sound, but we demonstrate that correlation among different parameters of a same system call improves the ability to model normality and detect outliers.

In SyscallAnomaly arguments are modeled according to their expected content. If the expected content is a file system path, the String Length, Character Distribution and Structural Inference models are used (collectively named "PathFactory"). If the expected content is a token (i.e., a flag, an opening mode, a UID or GID, and so on) the Token Search model is used instead ("FlagFactory"). A list of all the modeled system calls, along with the type of modeled values, is reported in Table I.

In [15], during the detection phase, the probability value for each call is obtained by computing the probability values for each

TABLE I
RECORDED SYSCALLS AND APPLIED MODELS IN SYSCALLANOMALY

SYSCALL NAME	MODEL APPLIED TO EACH ARGUMENT
`open`	`pathname` → PathFactory `flags` → FlagFactory `mode` → *none*
`execve`	`filename` → FlagFactory `argv` → PathFactory
`setuid, setgid`	`uid, gid` → FlagFactory
`setreuid, setregid`	`ruid, rgid, euid` → FlagFactory
`setresuid, setresgid`	`ruid, euid, suid` → FlagFactory
`rename` `symlink, link`	`oldpath, newpath` → PathFactory
`mount`	`source, target` → PathFactory `flags` → FlagFactory
`umount`	`target` → PathFactory `flags` → FlagFactory
`exit`	`status` → FlagFactory
`chown` `lchown`	`path` → PathFactory `owner, group` → FlagFactory
`chmod, mkdir` `creat`	`path` → PathFactory `mode` → FlagFactory
`mknode`	`pathname` → PathFactory `mode, dev` → FlagFactory
`unlink, mkdir`	`pathname` → PathFactory

of the models of each argument and then aggregating these models using $P(c) = \frac{\sum_{\forall m \in M} c_m * \log(p_m)}{|M|}$, where M is the set of stored models, c_m is the confidence and p_m is the model probability. On the other hand, in the extended version proposed in [34] the authors aggregate the values using a Bayesian network, showing an improvement in the detection rates. However, since in this work we focus on the improvement of the base models, and on the addition of time correlation, our work is pretty much independent of the anomaly score aggregation method. Additionally, the improved version of the software is not generally available for testing. Since a large number of non obvious configuration parameters (namely the Conditional Probability Table values) need to be properly chosen in order to replicate the results, and in [34] these values are not given, we chose to work on the original version of LibAnomaly/SyscallAnomaly, which is both simpler and readily available for download. The focus of our work, as already stated, is the improvement of models, the clustering of system calls, and the introduction of a Markovian model for time correlation. All of our improvements could be integrated with the proposed Bayesian framework in a very straightforward manner.

Not all system calls are modeled in these systems, nor in ours. Out of more than 280 syscalls implemented in Linux, only 22 are considered, because they are the only ones that are invoked enough times to generate significant profiles, yet are sufficiently characterized to generate meaningful models.

IV. BEYOND SYSCALLANOMALY: OUR PROPOSAL

Analyzing both the theoretical foundations described in [15], [34], and the results of our tests, in this paper we propose an alternative system, which implements some of the ideas of SyscallAnomaly along with clustering, Markovian based modeling, and behavior identification.

A. A constructive critique of SyscallAnomaly

In order to replicate the original tests of SyscallAnomaly, we used the host-based auditing data in BSM format contained in the (IDEVAL) dataset (which we describe more in depth in Section V-A). For now, it is sufficient to note that we used the BSM audit logs from the system named `pascal.eyrie.af.mil`, which runs a Solaris 2.5.1 operating system. The dataset contains 25 buffer overflow attacks against 4 different applications: `eject`, `fdformat`, `ps`, and `ffbconfig` (not tested). We used data from weeks 1 and 3 for training, and data from weeks 4 and 5 for testing the detection phase. However, it must be noted that some attacks are not directly detectable through system call analysis. The most interesting attacks for testing SyscallAnomaly are the ones in which an attacker exploits a vulnerability in a local or remote service to allow an intruder to obtain or escalate privileges.

TABLE II
EVALUATION OF SYSCALLANOMALY ON THE IDEVAL DATASET.

PROGRAM	FALSE POSITIVES	
	REPORTED IN [15]	OUR EXPERIMENT (#syscalls)
`fdformat`	0	1 (4)
`eject`	0	1 (6)
`ps`	0	2 (10)
`ftpd`	14	2 (45)
`telnetd`	17	2 (198)
`sendmail`	8	4 (97)

In addition to the programs named above, we ran SyscallAnomaly also on three other programs, namely `ftpd`, `sendmail` and `telnetd`, which are known not to be subject to any attack in the dataset, in order to better evaluate the false positive rate of the system. In Table II we compare our results with the released version of SyscallAnomaly [38] to reproduce the results reported in [15].

As can be seen, our results are different from those reported in [15], but the discrepancy can be explained by a number of factors:

- the version of SyscallAnomaly and LibAnomaly available online could be different from or older than the one used for the published tests;
- several parameters can be tuned in SyscallAnomaly, and a different tuning could produce different results;
- part of the data in the IDEVAL dataset under consideration are corrupted or malformed;
- in [15] it is unclear if the number of false positives is based on the number of executions erroneously flagged as anomalous, or on the number of anomalous syscalls detected.

These discrepancies make a direct comparison difficult, but our numbers confirm that Syscall Anomaly performs well overall as a detector.

Studying in detail each false and true positive we were able to understand how and where SyscallAnomaly fails, and to devise several improvements over it. To give a brief example of the process we went through, let us consider `eject`: it is a very plain, short program, used to eject removable media on UNIX-like systems: it has a very simple and predictable execution flow, and thus it is straightforward to characterize: dynamic libraries are loaded, the device `vol@0:volctl` is accessed, and finally, the device `unnamed_floppy` is accessed.

The dataset contains only one kind of attack against `eject`, a buffer overflow with command execution (see Table III). The exploit is evident in the `execve` system call, since the buffer overflow is exploited from the command line. Many of the models in SyscallAnomaly are able to detect this problem: the character distribution model, for instance, performs quite well. The anomaly

TABLE III

A TRUE POSITIVE AND A FALSE POSITIVE ON EJECT.

TRUE POSITIVE

System Call	execve	
filename	/usr/bin/eject	
argv	eject\0x20\0x20\0x20\0x20[...]	
Argument	**Model**	**Prob. (Conf.)**
filename	Token Search	0.999999 (0)
argv	String Length	10^{-6} (0)
	Character Distribution	0.005 (0.928)
	Structural Inference	10^{-6} (0.025)
Total Score (Threshold)		*1.316 (0.0012)*

FALSE POSITIVE

System Call	open	
pathname	/vol/dev/rdiskette0/c0t6d0/volume_1	
flags	crw-rw-rw-	
Argument	**Model**	**Prob. (Conf.)**
pathname	String Length	0.667 (0.005)
	Character Distribution	0.99 (0.995)
	Structural Inference	10^{-6} (1)
flags	Token Search	0.999 (1)
Total Score (Threshold)		*8.186 (1.454)*

TABLE IV

TRUE POSITIVE ON FDFORMAT: OPENING LOCALIZATION FILE

System Call	open	
pathname	/usr/lib/locale/iso_8859_1/[...]	
flags	-r-xr-xr-x	
Argument	**Model**	**Prob. (Conf.)**
pathname	String Length	0.0096 (0.005)
	Character Distribution	0.005 (0.995)
	Structural Inference	10^{-6} (0.986)
flags	Token Search	10^{-6} (1)
Total Score (Threshold)		*8.186 (1.454)*

value turns out to be 1.316, much higher than the threshold (0.0012). The String Length and Structural Inference models flag this anomaly as well, but interestingly they are mostly ignored since their confidence value is too low. The confidence value for the TokenSearch model is 0, which in SyscallAnomaly convention means that the field is not recognized as a token. This is actually a shortcoming of the association of models with parameters in SyscallAnomaly, because the "filename" argument is not really a token.

A false positive happens when a removable unit, unseen during training, is opened (see Table III). The Structural Inference model is the culprit of the false alert, since the name structure is different from the previous one for the presence of an underscore. As we will see later on, the extreme brittleness of the transformation and simplification model is the main weakness of the Structural Inference model.

Another alert happens in the opening of a localization file (Table IV), which triggers the string length model and creates an anomalous distribution of characters; moreover, the presence of numbers, underscores and capitals creates a structure that is flagged as anomalous by the Structural Inference model. The anomaly in the Token Search model is due to the fact that the open mode (-r-xr-xr-x) is not present in any of the training files. This is not an attack, but is the consequence of the buffer overflow attack, and as such is counted as a true positive. However, it is more likely to be a lucky, random side effect.

Without getting into similar details for all the other programs we analyzed (details which can be found in [39]), let us summarize our findings. ps is a jack-of-all-trades program to monitor process execution, and as such is much more articulated in its options and execution flow than any of the previously analyzed executables. However, the sequence of system calls does not vary dramatically depending on the user specified options: besides library loading, the program opens /tmp/ps_data and the files containing process information in /proc. Also in this case, attacks are buffer overflows on a command-line parameter. In this case, as was the case for fdformat, a correlated event is also detected, the opening of file /tmp/foo instead of file /tmp/ps_data. Both the Token Search model and the Structural Inference model flag an anomaly, because the opening mode is unseen before, and because the presence of an underscore in /tmp/ps_data makes it structurally different from /tmp/foo. However, if we modify the exploit to use /tmp/foo_data, the Structural Inference model goes quiet. A false positive happens when ps is executed with options lux, because the Structural Inference model finds this usage of parameters very different from -lux (with a dash), and therefore strongly believes this to be an attack. Another false positive happens when a zone file is opened, because during training no files in zoneinfo were opened. In this case it is very evident that the detection of the opening of the /tmp/foo file is more of another random side effect than a detection, and in fact the model which correctly identifies it also creates false positives for many other instances.

In the case of in.ftpd, a common FTP server, a variety of commands could be expected. However, because of the shortcomings of the IDEVAL dataset (see Section V-A below), the system call flow is fairly regular. After access to libraries and configuration files, the logon events are recorded into system log files, and a vfork call is then executed to create a child process to actually serve the client requests. In this case, the false positives mostly happen because of the opening of files never accessed during training, or with "unusual modes".

sendmail is a really complex program, with complex execution flows that include opening libraries and configuration files, accessing the mail queue (/var/spool/mqueue), transmitting data through the network and/or saving mails on disk. Temporary files are used, and the setuid call is also used, with an argument set to the recipient of the message (for delivery to local users). A false positive happens for instance when sendmail uses UID 2133 to deliver a message. In training that particular UID was not used, so the model flags it as anomalous. Since this can happen in the normal behavior of the system, it is evidently a generic problem with the modeling of UIDs as it is done in LibAnomaly. Operations in /var/mail are flagged as anomalous because the file names are similar to /var/mail/emonca000Sh, and thus the alternation of lower and upper case characters and numbers easily triggers the Structural Inference model.

We outlined different cases of failure of SyscallAnomaly. But what are the underlying reasons for these failures? The **Structural Inference model** turns out to be the weakest one. It is too sensitive against non alphanumeric characters, since they are not altered nor compressed: therefore, it reacts strongly against slight modifications that involve these characters. This becomes visible when libraries with variable names are opened, as it is evident in the false positives generated on the ps program. On the other hand, the compressions and simplifications introduced

TABLE V
BEHAVIOR OF SYSCALLANOMALY WITH AND WITHOUT THE STRUCTURAL INFERENCE MODEL

Program	False Positives (Syscalls flagged)	
	With the HMM	Without the HMM
fdformat	1 (4)	1(4)
eject	1 (6)	1 (3)
ps	2 (10)	1 (6)
ftpd	2 (45)	2 (45)
telnetd	2 (198)	0 (0)
sendmail	4 (97)	4 (97)

TABLE VI
PERCENTAGE OF `OPEN` SYSCALLS AND NUMBER OF EXECUTIONS (PER PROGRAM) IN THE IDEVAL DATASET.

PROGRAM	% OF open	#EXECUTIONS
fdformat	92.42%	5
eject	93.23%	7
ps	93.62%	105
telnetd	91.10%	65
ftpd	95.66%	1082
sendmail	86.49%	827

are excessive, and cancel out any interesting feature: for instance, the strings `/tmp/tempfilename` and `/etc/shadow` are indistinguishable by the model. Another very surprising thing, as we already noticed, is the choice of ignoring the probability values in the Markov model, turning it into a binary value (0 if the string cannot be generated, 1 otherwise). This assumes an excessive weight in the total probability value, easily causing a false alarm. To verify our intuition, we re-ran the tests excluding the Structural Inference model: the detection rate is unchanged, while the false positive rate strongly diminishes, as shown in Table V (once again, both in terms of global number of alerts, and of flagged system calls). Therefore, the Structural Inference model is not contributing to detection; instead it is just causing a growth in the anomaly scores which could lead to an increased number of false positives. The case of `telnetd` is particularly striking: excluding the Structural Inference model makes all the false positives disappear.

The **Character Distribution model** is much more reliable, and contributes positively to detection. However, it is not accurate about the particular distribution of each character, and this can lead to possible mimicry attacks. For instance, executing `ps -[x)` has a very high probability, because it is indistinguishable from the usual form of the command `ps -axu`.

The **Token Search model** has various flaws. First of all, it is not probabilistic, as it does not consider the relative probability of the different values. Therefore, a token with 1000 occurrences is considered just as likely as one with a single occurrence in the whole training set. This makes the training phase not robust against the presence of outliers or attacks in the training dataset. Additionally, since the model is applied only to fields that have been determined beforehand to contain a token, the statistical test is not useful: in fact, in all our experiments, it never had a single negative result. It is also noteworthy that the actual implementation of this test in [38] differs from what is documented in [15], [34], [37].

Finally, the **String Length model** works very well, even if this is in part due to the artifacts in the dataset, as we describe in Section V-A.

B. Improving SyscallAnomaly

We can identify and propose three key improvements over SyscallAnomaly. Firstly, we redesign improved models for anomaly detection on arguments, focusing on their reliability. Over these improved models, we introduce a clustering phase to create correlations among the various models on different arguments of the same syscall: basically, we divide the set of the invocations of a single system call into subsets which have arguments with an higher similarity. This idea arises from the consideration that some system calls do not exhibit a single normal behavior, but a plurality of behaviors (ways of use) in different portions of a program. For instance, as we will see in the next sections, an `open` syscall can have a very different set of arguments when used to load a shared library or a user-supplied file. Therefore, the clustering step aims to capture relationships among the values of various arguments (e.g. to create correlations among some filenames and specific opening modes). In this way we can achieve better characterization.

Finally, we introduce a sequence-based correlation model through a Markov chain. This enables the system to detect deviations in the control flow of a program, as well as abnormalities in each individual call, making evident the whole anomalous context that arises as a consequence, not just the single point of an attack. The combination of these improvements solves the problems we outlined in the previous sections, and the resulting prototype outperforms SyscallAnomaly, achieving also a better generality.

C. Clustering of system calls

We applied a hierarchical agglomerative clustering algorithm to find, for each system call, subclusters of invocation with similar arguments; we are interested in creating models on these clusters, and not on the general system call, in order to better capture normality and deviations. This is important because, as can be seen from Table VI, in the IDEVAL dataset the single system call `open` constitutes up to 95% of the calls performed by a process. Indeed, `open` is probably the most used system call on UNIX-like systems, since it opens a file or device in the file system and creates a handle (descriptor) for further use. `open` has three parameters: the file path, a set of flags indicating the type of operation (e.g. read-only, read-write, append, create if non existing, etc.), and an optional opening mode, which specifies the permissions to set in case the file is created. Only by careful aggregation over these parameters we may divide each "polyfunctional" system call into "subgroups" that are specific to a single functionality.

We used a single-linkage, bottom-up agglomerative technique. Conceptually, such an algorithm initially assigns each of the N input elements to a different cluster, and computes an $N \times N$ distance matrix D. Distances among clusters are computed as the *minimum* distance between an element of the first cluster and an element of the second cluster. The algorithm progressively joins the elements i and j such that $D(i,j) = \min(D)$. D is updated by substituting i and j rows and columns with the row and column of the distances between the newly joined cluster and the remaining ones. The minimum distance between two different clusters $d_{stop,min}$ is used as a stop criterion, in order to

TABLE VII
EXECUTION TIMES WITH AND WITHOUT THE HEURISTIC (AND IN PARENTHESIS, VALUES OBTAINED BY PERFORMANCE TWEAKS)

PROGRAM	#ELEMENTS	NAÏVE	OPTIMIZED
`fdformat`	11	0.14" (0.12")	0.014" (0.002")
`eject`	13	0.24" (0.13")	0.019" (0.003")
`ps`	880	19'52" (37")	7" (5")
`sendmail`	3450	unable to complete	7'19" (6'30")

TABLE VIII
ASSOCIATION OF MODELS TO SYSCALL ARGUMENTS IN OUR PROTOTYPE

SYSCALL	MODEL USED FOR THE ARGUMENTS
`open`	`pathname` → Path Name `flags, mode` → Discrete Numeric
`execve`	`filename` → Path Name `argv` → Execution Argument
`setuid, setgid`	`uid, gid` → User/Group
`setreuid, setregid`	`ruid, euid` → User/Group
`setresuid, setresgid`	`ruid, euid, suid` → User/Group
`symlink, link, rename`	`oldpath,newpath` → Path Name
`mount`	`source, target` → Path Name `flags` → Discrete Numeric
`umount`	`target, flags` → Path Name
`exit`	`status` → Discrete Numeric
`chown` `lchown`	`path` → Path Name `group, owner` → User/Group
`chmod, mkdir` `creat`	`path` → Path Name `mode` → Discrete Numeric
`mknode`	`pathname` → Path Name `mode, dev` → Discrete Numeric
`unlink, rmdir`	`pathname` → Path Name

prevent the clustering process from lumping all of the system calls together; moreover, a lower bound $d_{stop,num}$ for the number of final clusters is used as a stop criterion as well. If any of the stop criteria is satisfied, the process is stopped. The time complexity of a naive implementation is roughly $O(N^2)$. This would be too heavy, in both time and memory. Besides introducing various tricks to speed up our code and reduce memory occupation (as suggested in [40]), we introduced an heuristic to reduce the average number of steps required by the algorithm. Basically, at each step, instead of joining just the elements at minimum distance d_{min}, also all the elements that are at a distance $d < \beta d_{min}$ from both the elements at minimum distance are joined, where β is a parameter of the algorithm. In this way, groups of elements that are very close together are joined in a single step, making the algorithm (on average) much faster, even if worst-case complexity is unaffected. Table VII indicates the results measured in the case of $d_{stop,min} = 1$ and $d_{stop,num} = 10$; we want to recall that these timings, seemingly very high, refer to the training phase and not to the run time phase.

The core step in creating a good clustering is of course the definition of the *distance* among different sets of arguments. We proceed by comparing corresponding arguments in the calls, and for each couple of arguments a we compute:

$$d_a = \begin{cases} K_{(\cdot)} + \alpha_{(\cdot)}\delta_{(\cdot)} & \text{if the elements are different} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $K_{(\cdot)}$ is a fixed quantity which creates a "step" between different elements, while the second term is the real distance between the arguments $\delta_{(\cdot)}$, normalized by a parameter $\alpha_{(\cdot)}$. Note that, the above formula is a template: the use of "$(\cdot)$" denotes that such variables are parametric w.r.t. the type of argument; how $K_{(\cdot)}$, $\alpha_{(\cdot)}$, and $\delta_{(\cdot)}$ are computed will be detailed below for each type of argument. The distance between two different system calls, i and j, is computed as the sum of distances among corresponding arguments $D(i,j) = \sum_{a \in \mathrm{A}} d_a$ (where A is the set of system call arguments).

Hierarchical clustering, however, creates a problem for the detection phase, since there is no concept similar to the "centroid" of partitioning algorithms that can be used for classifying new inputs, and re-clustering the whole dataset after each new input is computationally unfeasible. Thus, we need to generate, from each cluster, a "representative model" that can be used to cluster (or classify) further inputs. This is a well known problem which needs a creative solution. For each type of argument, we decided to develop a stochastic model that can be used to this end.

These models should be able to associate a *probability* to inputs, i.e., to generate a *probability density function* that can be used to state the probability with which the input belongs to the model. As we will see, in most cases, this will be in the form of a discrete probability, but more complex models such as HMMs will also be used. Moreover, a concept of distance must be defined between each model and the input. The model should be able to "incorporate" new candidates during training, and to slowly adapt in order to represent the whole cluster. It is important to note that it is not strictly necessary for the candidate model and its distance functions to be the same used for clustering purposes. It is also important to note that the clustering could be influenced by the presence of outliers (such as attacks) in the training set. This could lead to the formation of small clusters of anomalous call instances. As we will see in Section IV-D, this does not inficiate the ability of the overall system to detect anomalies.

As previously stated, at least 4 different types of arguments are passed to system calls: path names and file names, discrete numeric values, arguments passed to programs for execution, users and group identifiers (`UID`s and `GID`s). For each type of argument, we designed a representative model and an appropriate distance function. In Table VIII we summarize the association of the models described above with the arguments of each of the system calls we take into account.

Path names and file names are very frequently used in system calls. They are complex structures, rich of useful information, and therefore difficult to model properly. A first interesting information is commonality of the *path*, since files residing in the same branch of the file system are usually more similar than the ones in different branches. Usually, inside a path, the *first* and the *last* directory carry the most significance. If the filename has a similar *structure* to other filenames, this is indicative: for instance common *prefixes* in the filename, such as the prefix `lib`, or common suffixes such as the extensions.

For the clustering phase, we chose to re-use a very simple model already present in SyscallAnomaly, the directory tree depth. This is easy to compute, and experimentally leads to fairly good results even if very simple. Thus, in Equation 1 we set δ_a to be the distance in depth. E.g.: let $K_{path} = 5$ and $\alpha_{path} = 1$; comparing `/usr/lib/libc.so` and `/etc/passwd` we obtain $d_a = 5 + 1 * 1 = 6$, while comparing `/usr/lib/libc.so` and `/usr/lib/libelf.so.1` we obtain $d_a = 0$.

After clustering has been done, we represent the path name of the files of a cluster with a probabilistic tree

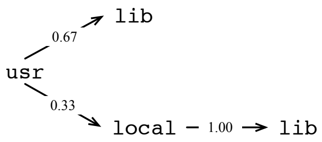


Fig. 1. Probabilistic tree example

which contains all the directories involved with a probability weight for each. For instance, if a cluster contains: `/usr/lib/libc.so.1`, `/usr/lib/libelf.so.1`, `/usr/local/lib/libintl.so.1`, the generated tree will be as in Fig. 1.

Filenames are usually too variable, in the context of a single cluster, to allow a meaningful model to be always created. However, we chose to set up a system-wide threshold below which the filenames are so regular that they can be considered a model, and thus any other filename can be considered an anomaly. The probability returned by the model is therefore $P_T = P_t * P_f$, where P_t is the probability that the path has been generated by the probabilistic tree and P_f is set to 1 if the filename model is not significant (or if it is significant and the filename belongs to the learned set), and to 0 if the model is significant and the filename is outside the set.

Discrete numeric values such as flags, opening modes, etc. are usually chosen from a limited set. Therefore we can store all of them along with a discrete probability. Since in this case two values can only be "equal" or "different", we set up a binary distance model for clustering, where the distance between x and y is:

$$d_a = \begin{cases} K_{disc} & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

and K_{disc}, as usual, is a configuration parameter. In this case, models fusion and incorporation of new elements are straightforward, as well as the generation of probability for a new input to belong to the model.

We also noticed that *execution argument* (i.e. the arguments passed to the `execve` syscall) are difficult to model, but we found the length to be an extremely effective indicator of similarity of use. Therefore we set up a binary distance model, where the distance between x and y is:

$$d_a = \begin{cases} K_{arg} & \text{if } |x| \neq |y| \\ 0 & \text{if } |x| = |y| \end{cases}$$

denoting with $|x|$ the length of x and with K_{arg} a configuration parameter. In this way, arguments with the same length are clustered together. For each cluster, we compute the minimum and maximum value of the length of arguments. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if its length belongs to the interval, and 0 otherwise.

Many arguments express `UID`s or `GID`s, so we developed an ad-hoc model for *users and groups* identifiers. Our reasoning is 3that all these discrete values have three different meanings: `UID` 0 is reserved to the super-user, low values usually are for system special users, while real users have `UID`s and `GID`s above a threshold (usually 1000). So, we divided the input space in these three groups, and computed the distance for clustering using the following formula:

$$d_a = \begin{cases} K_{uid} & \text{if belonging to different groups} \\ 0 & \text{if belonging to the same group} \end{cases}$$

and K_{uid}, as usual, is a user-defined parameter. Since `UID`s are limited in number, they are preserved for testing, without associating a discrete probability to them. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if the UID belongs to the learned set, and 0 otherwise.

TABLE IX
CLUSTER VALIDATION PROCESS

PROGRAM	#ELEMENTS	% CORRECT ASSIGNMENTS	CONF.
`fdformat`	10	100%	1
`eject`	12	100%	1
`ps`	525	100%	1
`telnetd`	38	100%	0.954
`ftpd`	69	97.1%	0.675
`sendmail`	3211	100%	0.996

This model based clustering is somehow error prone since we would expect obtained centroids to be more general and thus somehow to interfere when clustering either new or old instances. To double check this possible issue we follow a simple process:

1) creation of clusters on the training dataset;
2) generation of models from each cluster;
3) use of models to classify the original dataset into clusters, and check that inputs are correctly assigned to the same cluster they contributed to create.

This is done both for checking the representativeness of the models, and to double-check that the different distances computed make sense and separate between different clusters. Table IX shows, for each program in the IDEVAL dataset (considering the representative open system call), the percentage of inputs correctly classified, and a confidence value, computed as the average "probability to belong" computed for each element w.r.t. the cluster it helped to build. The results are almost perfect, as expected, with a lower value for the ftpd program, which has a wider variability in filenames.

D. Characterizing process behavior

In order to take into account the execution context of each system call, we use a first order Markov chain to represent the program flow. The model states represent the system calls, or better they represent the various clusters of each system call, as detected during the clustering process. For instance, if we detected three clusters in the `open` syscall, and two in the `execve` syscall, then the model will be constituted by five states: $\texttt{open}_1$, $\texttt{open}_2$, $\texttt{open}_3$, $\texttt{execve}_1$, $\texttt{execve}_2$. Each transition will reflect the probability of passing from one of these groups to another through the program. As we already observed in Section II, this approach was investigated in former literature, but never in conjunction with the handling of parameters and with a clustering of system calls based on such parameters.

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

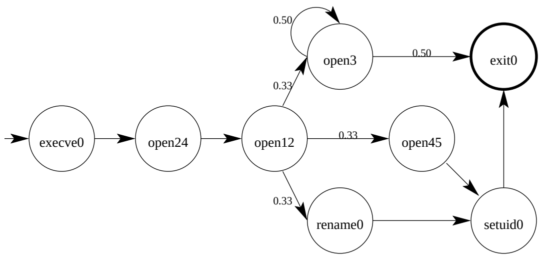


Fig. 2. Example of Markov model (transition probability is 1.00 unless specified).

composite probability along all known models: $\max(\prod_{i\in \mathrm{M}} P_i)$. The probabilities of the Markov model are then straightforward to compute. The final results can be similar to what is shown in Figure 2. On a first sight, this could resemble a simple Markov chain, however it should be noticed that each of the state of this Markov model could be mapped into the set of possible cluster elements associated to it. From this perspective, it could be seen as a very specific HMM where states are fully observable through an uniform emission probability over the associated syscalls. By simply turning the clustering algorithm into one with overlapping clusters we could obtain a proper HMM description of the user behavior, as it would be if we decide to further merge states together. This is actually our ongoing work toward full HMM behavior modeling with the aim, through overlapping syscalls clustering, of improving the performance of the classical Baum-Welch algorithm and solving the issue of HMM hidden space cardinality selection [41]. From our experiments, in the case of the simple traces like the ones found in the IDEVAL dataset, the simpler model we use is sufficient, so we are working on obtaining more complex and realistic scenarios before attempting to improve the proposed algorithm.

This type of model 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 then an attack-free training set.

For detection, three distinct probabilities can be computed for each executed syscall: the probability of the *execution* sequence inside the Markov model up to now, P_s; the probability of the *syscall* to belong to the best-matching cluster, P_c; the *last transition* probability in the Markov model, P_m.

The latter two probabilities can be combined into a probability of the single syscall, $P_p = P_c \cdot P_m$, keeping a separate value for the "sequence" probability P_s. In order to set appropriate threshold values, we use the training data, compute the lowest probability over all the dataset for that single program (both for the sequence probability and for the punctual probability), and set this (eventually modified by a tolerance value) as the anomaly threshold. The tolerance can be tuned to trade off detection rate for false positive rate.

During detection, each system call is considered in the context of the process. The cluster models are once again used to classify each syscall into the correct cluster as explained above: therefore $P_c = \max(\prod_{i\in \mathrm{M}} P_i)$, where P_s and P_m are computed from the Markov model, and require our system to keep track of the current state for each running process. If either P_s or $P_p = P_c \cdot P_m$ are lower than the anomaly threshold, the process is flagged as malicious.

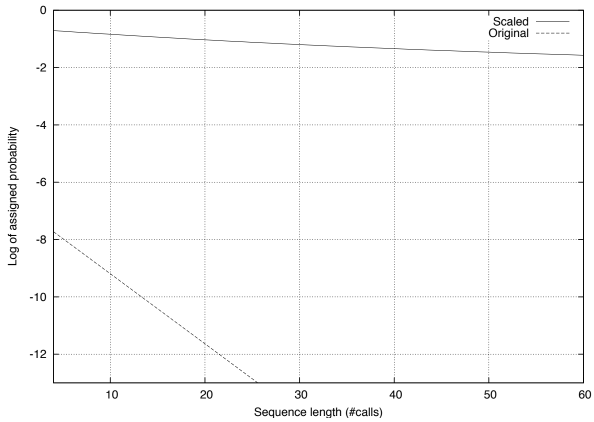


Fig. 3. Measured sequence probability (log) vs. sequence length, comparing the original calculation and the second variant of scaling.

It is important to note that, given an l-long sequence of system calls, its sequence probability is $P_s(l) = \prod_{i=0}^{l} P_p(i)$ where $P_p(i) \in [0,1]$ is the probability of the i-th system call in the sequence. Therefore, it is self-evident that $\lim_{l\to+\infty} P_s(l) = 0$. Experimentally, we observed that the sequence probability quickly decreases to zero, even for short sequences (on the IDEVAL dataset, we found that $P_s(l) \simeq 0$ for $l \geq 25$). This leads to a high number of false positives, since many sequences are assigned probabilities close to zero (thus, always lower than any threshold value).

To overcome this shortcoming, we implemented two "scalings" of the probability calculation, both based on the geometric mean. As a first attempt, we computed $P_s(l) = \sqrt[l]{\prod_{i=1}^{l} P_p(i)}$, but in this case $\mathrm{P}\left[\lim_{l\to+\infty} P_s(l) = e^{-1}\right] = 1$.

Proof: Let $\mathrm{G}(P_p(1), \ldots, P_p(l)) = \sqrt[l]{\prod_{i=1}^{l} P_p(i)}$ the geometric mean of the sample $P_p(1), \ldots, P_p(l)$. If we assume that the sample is generated by a uniform distribution (i.e., $P_p(1), \ldots, P_p(l) \sim \mathcal{U}(0,1)$) then $\log P_p(i) \sim \mathcal{E}(\beta = 1)\ \forall i = 1, \ldots, l$: this can be proven by observing that the *Cumulative Distribution Function* (CDF) [42] of $\log X$ (with $X = P_p \sim \mathcal{U}(0,1)$) equals the CDF of an exponentially distributed variable with $\beta = 1$.

The arithmetic mean $\mathrm{A}(\cdot)$ of the sample $-\log(P_p(1)), \ldots, -\log(P_p(l))$ converges (in probability) to $\beta = 1$ for $l \to +\infty$, that is:

$$\mathrm{P}\left[\lim_{l\to+\infty} \frac{1}{l}\sum_{i=1}^{l} -\log P_p(i) = -\beta = -1\right] = 1$$

because of the strong law of large numbers.

Being the geometric mean $\mathrm{G}(P_p(1), \ldots, P_p(l)) = \left(\prod_{i=1}^{l} P_p(i)\right)^{\frac{1}{l}} = e^{\frac{1}{l}\sum_{i=1}^{l} -\log P_p(i)}$, we have

$$\mathrm{P}\left[\lim_{l\to+\infty}\left(e^{\frac{1}{l}\sum_{i=1}^{l} -\log P_p(i)}\right) = e^{-\beta} = e^{-1}\right] = 1$$

■

This is not our desired result, so we modified this formula to introduce a sort of "forgetting factor": $P_s(l) = \sqrt[2l]{\prod_{i=1}^{l} P_p(i)^i}$. In this case, we can prove that $\mathrm{P}\left[\lim_{l\to+\infty} P_s(l) = 0\right] = 1$.

Proof: The convergence of $P_s(l)$ to zero can be proven by observing that:

$$
\begin{aligned}
P_s(l) &= \left(\sqrt[l]{\prod_{i=1}^{l} P_p(i)^i}\right)^{\frac{1}{2}} = \left(e^{\frac{1}{l}\sum_{i=1}^{l} i\cdot\log P_p(i)}\right)^{\frac{1}{2}} = \\
&= \left(e^{\frac{1}{l}\sum_{j=0}^{l}\left(\sum_{i=1}^{l-j}\log P_p(i)\right)}\right)^{\frac{1}{2}} = \\
&= \left(e^{\sum_{j=0}^{l}\left(\frac{1}{l}\sum_{i=1}^{l}\log P_p(i)-\frac{1}{l}\sum_{i=l-j+1}^{l}\log P_p(i)\right)}\right)^{\frac{1}{2}}
\end{aligned}
$$

Because of the previous proof, we can write that:

$$\mathrm{P}\left[\lim_{l\to+\infty}\frac{1}{l}\sum_{i=1}^{l}\log P_p(i) = -1\right] = 1$$

We can further observe that, being $\log P_p(i) < 0$:

$$\forall j,l>0:\sum_{i=1}^{l}\log P_p(i) < \sum_{i=l-j+1}^{l}\log P_p(i)$$

therefore, the exponent is a sum of infinite negative quantities lesser than 0, leading us to the result that, *in probability*

$$
\begin{aligned}
&\lim_{l\to+\infty}\left(e^{\sum_{j=0}^{l}\left(\frac{1}{l}\sum_{i=1}^{l}\log P_p(i)-\frac{1}{l}\sum_{i=l-j+1}^{l}\log P_p(i)\right)}\right)^{\frac{1}{2}} = \\
&\lim_{x\to-\infty} e^x = 0
\end{aligned}
$$

■

Even if this second variant once again makes $P_s(l) \to 0$ (in probability), our experiments have shown that this effect is much *slower* than in the original formula: $P_s(l) \simeq 0$ for $l \geq 300$ (vs. $l \geq 25$ of the previous version), as shown in Fig. 3. In fact, this scaling function also leads to much better results in terms of false positive rate (see Section V).

A possible alternative, which we are currently exploring, is the exploitation of distance metrics between Markov models [35], [36]. to define robust criteria for comparing new and learned sequence models. Basically, the idea is to create and continuously update a Markov model associated to the program instance being monitored, and to check how much such a model differs from the ones the system has learned for the same program. This approach is complementary to the one proposed above, since it requires long sequences to get a proper Markov model. So, the use of both criteria (sequence likelihood in short activations, and model comparison in longer ones) could lead to a reduction of false positives on the sequence model.

E. Prototype implementation

We implemented the above described system into a two-stage, highly configurable, modular architecture written in ANSI C. The high-level structure is depicted in Figure 4: the Detection module implements the core IDS functionalities, Compressor is in charge of the clustering phase while BehaviorModeler implements Markov modeling features. Both Compressor and BehaviorModeler are used in both training phase and detection phase, as detailed below.

The Compressor module implements abstract clustering procedures along with abstract representation and storage of generated clusters. The BehaviorModeler is conceptually similar to Compressor: it has a basic Markov chains implementation, along with ancillary modules for model handling and storage.

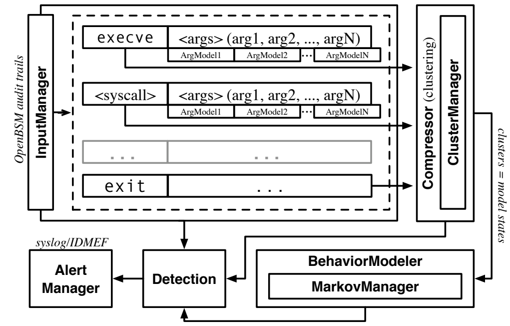


Fig. 4. The high-level structure of our prototype.

During training, Compressor is invoked to create the clusters on a given system call trail while BehaviorModeler infers the Markov model according to the clustering. At running time, for each system call, Detection uses ClusterManager to find the appropriate cluster; it also invokes MarkovManager to follow the actual behavior of the monitored process: if significant deviations are found, the AlertManager is triggered and alarms are fired and logged accordingly.

The AlertManager can output to both standard output, syslog facilities and IDMEF files. Both the clustering phase and the behavioral analysis are multithreaded and intermediate results of both procedures can be dumped in XML.

V. Result analysis

In this section, we both compare the detection accuracy of our proposal and analyze the performances of the running prototype we developed. Because of the known issues of IDEVAL (plus our findings reported in the following), we also collected fresh training data and new attacks to further prove that our proposal is promising in terms of accuracy.

A. Regularities in host-data of IDEVAL

A well-known problem in IDS research is the lack of reliable sources of test data. The "DARPA IDS Evaluation dataset" or IDEVAL is basically the only dataset of this kind which is freely available along with truth files; in particular we used the 1999 dataset [43]. These data are artificially generated and contain both network and host auditing data. A common question is how realistic these data are. Many authors already analyzed the network data of the 1999 dataset, finding many shortcomings [44], [45]. Our own analysis [39] of the 1999 host-based auditing data revealed that this part of the dataset is all but immune from problems. The first problem is that in the training datasets there are too few execution instances for each software, in order to properly model its behavior, as can be seen in Table VI. Out of (just) 6 programs present, for two (`fdformat` and `eject`), only a handful of executions is available, making training unrealistically simple.

The number of system calls used is also extremely limited, making execution flows very plain. Additionally, most of these executions are similar, not covering the full range of possible execution paths of the programs (thus causing overfitting of any anomaly model). For instance, in Fig. 5 we have plotted the frequency of the length (in system calls) of the various executions of `telnetd` on the training data. The natural clustering of the

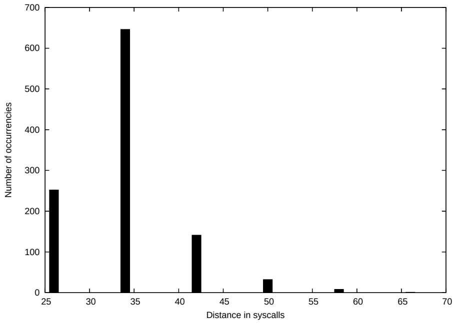


Fig. 5. `telnetd`: distribution of the number of other system calls among two `execve` system calls (i.e., distance between two consecutive `execve`).

data in a few groups clearly shows how the executions of the program are sequentially generated with some script, and suffer of a lack of generality.

System calls arguments show the same lack of variability: in all the training dataset, all the arguments of the system calls related to `telnetd` belong to the following set:

`fork, .so.1, utmp, wtmp, initpipe, exec, netconfig, service_door, :zero, logindmux, pts`

The application layer contains many flaws, too. For instance, the FTP operations (30 sessions on the whole) use a very limited subset of file (on average 2 per session), and are performed always by the same users on the same files, for a limitation of the synthetic generator of these operations. In addition, during training, no uploads or idle sessions were performed. Command executions are also highly predictable: for instance, one script always execute a cycle composed of `cat`, `mail`, `mail` again, and at times `lynx`, sometimes repeated twice. The same happens (but in a random order) for `rm`, `sh`, `ps` and `ls`. In addition, a number of processes have evidently crafted names (e.g. `logout` is sometimes renamed `lockout` or `log0ut`); the same thing happens with path names, which are sometimes different (e.g. `/usr/bin/lynx` or `/opt/local/bin/lynx`), but an analysis shows that they are the same programs (perhaps symbolic links generated to create noise over the data). The combination of the two creates interesting results such as `/etc/loKout` or `/opt/local/bin/l0gout`. In a number of cases, processes `lynx`, `mail` and `q` have duplicate executions with identical PID and timestamps, and with different paths and/or different arguments; this is evidently an inexplicable flaw of the dataset. We also found many program executions to be curiously meaningless. In fact, the BSM traces of some processes contain just `execve` calls, and this happens for 28% of the programs in the testing portion dataset (especially for those with a crafted name, like `loKout`). It is obvious that testing an host-based IDS with one-syscall-long sequences does not make a lot of sense, not to talk about the relevance of training against such sequences.

An additional problem is that since 1999, when this dataset was created, everything changed: the usage of network protocols, the protocols themselves, the operating systems and applications used. For instance, all the machines involved are Solaris version 2.5.1 hosts, which are evidently ancient nowadays. The attacks are similarly outdated: the only attack technique used are buffer overflows, and all the instances are detectable in the `execve` system call arguments. Nowadays attackers and attack type are much more complex than this, operating at various layers of the network and application stack, with a wide range of techniques and scenarios that were just not imaginable in 1999.

To give an idea of this, we were able to create a detector which finds all the buffer overflow attacks without any false positive: a simple script which flags as anomalous any argument longer than 500 characters can do this (because all the overflows occur in the parsing of the command line, which is part of the parameters of the `execve` system call which originates the process). This is obviously unrealistic.

Other datasets exist (e.g. the DEFCON CTF packet captures [46]), but they are not labeled and do not contain "background traffic". Thus, most existing researches on network-based IDSs use the DARPA datasets for evaluation. This is a crucial factor: any bias or error in the DARPA dataset has influenced, and will influence in the future, the very basic research on this topic.

B. Experimental setup

In order to avoid such shortcomings, besides the use of IDEVAL for comparison purposes with SyscallAnomaly (which was tested on that dataset), we generated an additional experimental dataset for two frequently used console applications (i.e., `bsdtar` and `eject`). We chose two different buffer overflow exploits that allow to execute arbitrary code. The exploit for the vulnerability of `mcweject 0.9` is public (`http://www.milw0rm.com/exploits/3578`), while the exploit against `bsdtar` was created by us and is based on a publicly disclosed vulnerability in the PAX handling function of `libarchive 2.2.3`, which basically does not check the length of the header of the parsed file, which is stored in a header field, resulting in an heap overflow which allows code injection through the creation of a malformed archive. As we detailed in Section IV-C, our system can be tuned to avoid overfitting; in the current implementation, such parameters can be specified for *each* system call, thus in the following we report the bounds of variations instead of listing *all* the single values: $d_{stop,num} \in \{1, 2, 3\}$, $d_{stop,min} = \{6, 10, 20, 60\}$.

Our testing platform runs a vanilla installation of FreeBSD 6.2 on a x86 machine; the kernel has been recompiled enabling the appropriate auditing modules. Since our systems, and other host-based anomaly detectors [15], [34], accept input in the BSM format, the OpenBSM [47] auditing tools collection has been used for collecting audit trails. We have audited vulnerable releases of `eject` and `bsdtar`, namely: `mcweject 0.9` (which is an alternative to the BSD `eject`) and the version of `bsdtar` which is distributed with FreeBSD 6.2.

The `eject` executable has a small set of command line option and a very plain execution flow. For the simulation of a legitimate user, we simply chose different permutations of flags and different devices. For this executable, we manually generated 10 executions, which are remarkably similar (as expected).

Creating a dataset of normal activity for the `bsdtar` program is more challenging. It has a large set of command line options, and in general is more complex than `eject`. While the latter is generally called with an argument of `/dev/*`, the former can be invoked with any argument string, for instance `bsdtar cf myarchive.tar /first/path /second/random/path` is a perfectly legitimate command line. Using a process similar to the one used for creating the IDEVAL dataset, and in fact used also in other works such as [11], we prepared a shell script which embeds pseudo-random behaviors of an average user who creates or extracts archives. To simulate user activity, the script randomly creates random-sized, random-content files inside a snapshot of a real-world desktop filesystem. In the case of the simulation of super-user executions, these files are scattered around the system; in the case of a regular user, they are into that user's own home directory. Once the filesystem has been populated, the tool randomly walks around the directory tree and randomly creates TAR archives. Similarly, found archives are randomly expanded. The randomization takes also into account the different use of flags made by users: for instance, some users prefer to uncompress an archive using `tar xf archive.tar`, many others still use the dash `tar -xf archive.tar`, and so on.

In [17] a real web and ssh server logs were used for testing. While this approach yields interesting results, we did not follow it for three reasons. Firstly, in our country various legal concerns limit what can be logged on real-world servers. In second place, http and ssh are complex programs where understanding what is correctly identified and what is not would be difficult (as opposed to simply counting correct and false alerts). Finally, such a dataset would not be reliable because of the possibility of the presence of real attacks inside the collected logs (in addition to the attacks inserted manually for testing).

C. Detection accuracy

For the reasons outlined above in Section V-A, as well as for the uncertainty outlined in Section IV-A, we did not rely on purely numerical results on detection rate or false positive rates. Instead, we compared the results obtained by our software with the results of SyscallAnomaly in the terms of a set of case studies, comparing them singularly. What turned out is that our software has two main advantages over LibAnomaly:

- a better contextualization of anomalies, which lets the system detect whether a single syscall has been altered, or if a sequence of calls became anomalous consequently to a suspicious attack;
- a strong characterization of subgroups with closer and more reliable sub-models.

As an example of the first advantage, let us analyze again the program `fdformat`, which was already analyzed in Section IV-A. As can be seen from Table X, our system correctly flags `execve` as anomalous (due to an excessive length of input). It can be seen that P_m is 1 (the system call is the one we expected), but the models of the syscall are not matching, generating a very low P_c. The localization file opening is also flagged as anomalous for two reasons: scarce affinity with the model (because of the strange filename), and also erroneous transition between the open subgroups `open2` and `open10`. In the case of such an anomalous transition, thresholds are shown as "undefined" as this transition has never been observed in training. The attack effect (`chmod` and the change of permissions on `/export/home/elmoc/.cshrc`) and various intervening syscalls are also flagged as anomalous because the transition has never been observed ($P_m = 0$); while reviewing logs, this also helps us in understanding whether or not the buffer overflow attack has succeeded. A similar observation can be done on the execution of `chmod` on `/etc/shadow` ensuing an attack on `eject`.

In the case of `ps`, our system flags the `execve` system call, as usual, for excessive length of input. File `/tmp/foo` is also detected as anomalous argument for `open`. In `LibAnomaly`, this happened just because of the presence of an underscore, and was easy to bypass. In our case, `/tmp/foo` is compared against a sub-cluster of `open` which contains only the `/tmp/ps_data`, and therefore will flag as anomalous, with a very high confidence, any other name, even if structurally similar. A sequence of `chmod` syscalls which are executed inside directory `/home/secret` as a result of the attacks are also flagged as anomalous program flows.

TABLE X
`FDFORMAT`: ATTACK AND CONSEQUENCES

Anomalous Syscall	`execve0 (START ⇒ execve0)`
filename	`/usr/bin/fdformat`
argv	`fdformat\0x20\0x20\0x20\0x20[...]`
P_c	0.1
P_m	1
$\mathbf{P_p}$ (thresh.)	0.1 (1)
Anomalous Syscall	`open10 (open2 ⇒ open10)`
pathname	`/usr/lib/locale/iso_8859_1/[...]`
flags	`-r-xr-xr-x`
mode	`33133`
P_c	$5 * 10^{-4}$
P_m	0
$\mathbf{P_p}$ (thresh.)	0 (*undefined*)
Anomalous Syscall	`open11 (open10 ⇒ open11)`
pathname	`/devices/pseudo/vol@0:volctl`
flags	`crw-rw-rw-`
mode	`8630`
P_c	1
P_m	0
$\mathbf{P_p}$ (thresh.)	0 (*undefined*)
Anomalous Syscall	`chmod (open11 ⇒ chmod)`
pathname	`/devices/pseudo/vol@0:volctl`
flags	`crw-rw-rw-`
mode	`8630`
P_c	0.1
P_m	0
$\mathbf{P_p}$ (thresh.)	0 (*undefined*)
Anomalous Syscall	`exit0 (chmod ⇒ exit0)`
status	`0`
P_c	1
P_m	0
$\mathbf{P_p}$ (thresh.)	0 (*undefined*)

TABLE XI
DETECTION RATES AND FALSE POSITIVE RATES ON TWO TEST PROGRAMS, WITH (Y) AND WITHOUT (N) MARKOV MODELS.

Markov	DR	FPR	
	Seq.	Seq.	Call
`bsdtar`			
Y	100%	1.6%	0.1%
N	88%	1.6%	0.1%
`eject`			
Y	100%	0%	0%
N	0%	0%	0%

Limiting the scope to the detection accuracy of our system, we performed several experiments with both `eject` and `bsdtar`, and we summarize the results in Table XI. The prototype has been trained with ten different execution of `eject` and more than a hundred executions of `bsdtar`. We then audited eight instances of the activity of `eject` under attack, while for `bsdtar` we logged seven malicious executions. We report detection rates and false positive rates with (Y) and without (N) the use of Markov models, and we compute false positive rates using cross-validation through the data set (i.e., by training the algorithm on a subset of the dataset and subsequently testing the other part of the dataset). Note that, to better analyze the false positives, we accounted for

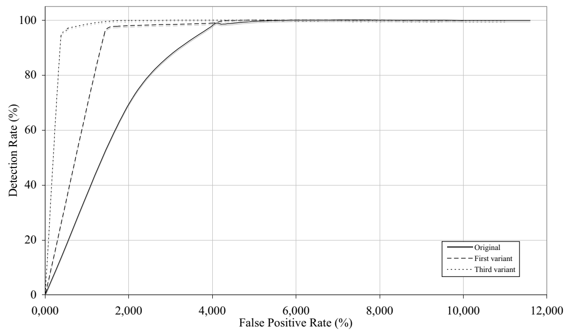


Fig. 6. Comparison of performance of different probability scaling functions.

TABLE XII
TRAINING AND DETECTION THROUGHPUT X.

TRAINING THROUGHPUT

Ses.	#Calls	#Progr.	t (Clust., Markov) [s]	X [$call/s$]
1	97644	111	12.056 (7.683, 3.268)	8099
2	34931	67	3.415 (1.692, 1.356)	10228
3	41133	129	6.721 (3.579, 2.677)	6120
4	50239	152	7.198 (3.019, 3.578)	6979
5	38291	115	4.503 (2.219, 1.849)	8503

DETECTION THROUGHPUT

Ses.	#Calls	#Progr.	t [s]	X [$call/s$]
1	109160	149	6.722	16239
2	160565	186	12.953	12395
3	103605	143	4.653	22266
4	115334	107	5.212	22128
5	112242	147	5.674	19781

both false positive sequences (Seq.) and false positive system calls (Call).

In both cases, using the complete algorithm yield a 100% detection rate with a very low false positive rate. In the case of `eject`, the exploit is detected in the very beginning: since a very long argument is passed to the `execve`, this triggers the argument model. The detection of the shellcode we injected exploiting the buffer overflow in `bsdtar` is identified by the `open` of the unexpected (special) file `/dev/tty`. Note that, the use of thresholds calculated on the overall Markov model allows us to achieve a 100% detection rate in the case of `eject`; without the Markov model, the attack would not be detected at all.

It is very difficult to compare our results directly with the other similar systems we identified in Section II. In [16] the evaluation is performed on the DARPA dataset, but detection rates and false positive rates are not given (the number of detections and false alarms is not normalized), so a direct comparison is difficult. Moreover, detection is computed using an arbitrary time window, and false alerts are instead given in "alerts per day". It is correspondingly difficult to compare against the results in [17], as the evaluation is ran over a dataset which is not disclosed, using two programs that are very different from the ones we use, and using a handful of exploits chosen by the authors. Different scalings of the false positives and detection rates also make a comparison impossible to draw.

As a side result, we tested the detection accuracy of the two scaling functions we proposed for computing the sequence probability P_s. As shown in Fig. 6, the first and the second variant both show lower false positive rate w.r.t. to the original, unscaled version.

D. Performance measurements

An IDS should not introduce significant performance overheads in terms of the time required to classify events as malicious (or not). An IDS based on the analysis of system calls has to intercept and process every single syscall invoked on the operating system by userspace applications; for this reason, the fastest a system call is processed, the best. We profiled the code of our system with `gprof` and `valgrind` for CPU and memory requirements. We ran the IDS on data drawn from the IDEVAL 1999 dataset (which is sufficient for performance measurements, as in this case we are only interested in the throughput and not in realistic detection rates).

In Table XII we reported the measurement of performance on the five working days of the first week of the dataset for training, and of the fourth week for testing. The throughput X varies during training between 6120 and 10228 syscalls per second. The clustering phase is the bottleneck in most cases, while the Markov model construction is generally faster. Due to the clustering step, the training phase is memory consuming: in the worst case, we recorded a memory usage of about 700 MB. The performance observed in the detection phase is of course even more important: in this case, it varies between 12395 and 22266 syscalls/sec. Considering that the kernel of a typical machine running services such as HTTP/FTP on average executes system calls in the order of thousands per second (e.g., around 2000 system calls per second for `wu-ftpd` [34]), the overhead introduced by our IDS is noticeable but does not severely impact system operations overall.

VI. CONCLUSIONS

In this paper we described a novel host-based IDS based on the analysis of system calls arguments and sequence. We analyzed previous literature on the subject, and found that there exists only a handful of works which take into account the anomalies in syscall arguments. We improved the models suggested in one of these works, we added a stage of clustering in order to characterize normal invocations of calls and to better fit models to arguments, and finally we complemented it with Markov models in order to capture correlation between system calls.

We outlined a number of new shortcomings in the IDEVAL dataset, demonstrating that (similarly to the known problems in the network data) the execution traces for system call analysis are too simple and predictable, not covering enough programs, nor exploring different types of executions. In addition, the dataset is hopelessly outdated, both in terms of attacks and of background operations. We outlined how we validated our results in order to obviate such glaring deficiencies of the dataset. We showed how the prototype is able to correctly contextualize alarms, giving the user more information to understand what caused any false positive, and to detect variations over the execution flow, as opposed to punctual variations over single instances. We also demonstrated its improved detection capabilities, and a reduction of false positives. The system is auto-tuning and fully unsupervised, even if a range of parameters can be set by the user to improve the quality of detection.

A possible future extension of this work is the analysis of complementary approaches (such as Markov model merging or the computation of distance metrics) to better detect anomalies in the case of long system call sequences, which we identified as a possible source of false positives.

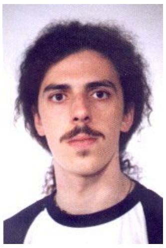
ACKNOWLEDGMENTS

We would like to thank, for a number of ideas, Davide Balzarotti, Giuliano Casale, William Robertson, and Prof. Ilenia Epifani. We also thank Prof. Giovanni Vigna for his comments on former drafts of this work. Davide Veneziano, Matteo Debiasi and Matteo Falsitta supported this work with software development and testing. We warmly thank Emanuele Oriano for proofreading, and the reviewers for their constructive comments.

REFERENCES

[1] J. P. Anderson, "Computer security threat monitoring and surveillance," J. P. Anderson Co., Ft. Washington, Pennsylvania, Tech. Rep., Apr 1980.
[2] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proc. of the 1996 IEEE Symp. on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1996.
[3] J. B. D. Cabrera, L. Lewis, and R. Mehara, "Detection and classification of intrusion and faults using sequences of system calls," *ACM SIGMOD Record*, vol. 30, no. 4, 2001.
[4] S. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. of Computer Security*, vol. 6, pp. 151–180, 1998.
[5] A. Somayaji and S. Forrest, "Automated response using system–call delays," in *Proc. of the 9th USENIX Security Symp.*, Denver, CO, Aug. 2000.
[6] W. W. Cohen, "Fast effective rule induction," in *Proc. of the 12th Int'l Conf. on Machine Learning*, A. Prieditis and S. Russell, Eds. Tahoe City, CA: Morgan Kaufmann, Jul 1995, pp. 115–123.
[7] W. Lee and S. Stolfo, "Data mining approaches for intrusion detection," in *Proc. of the 7th USENIX Security Symp.*, San Antonio, TX, 1998.
[8] D. Ourston, S. Matzner, W. Stump, and B. Hopkins, "Applications of Hidden Markov Models to detecting multi-stage network attacks," in *Proc. 36th Ann. Hawaii Int'l Conf. on System Sciences*, 2003, p. 334.
[9] S. Jha, K. Tan, and R. A. Maxion, "Markov chains, classifiers, and intrusion detection," in *CSFW '01: Proc. of the 14th IEEE Workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 2001, p. 206.
[10] C. C. Michael and A. Ghosh, "Simple, state-based approaches to program-based anomaly detection," *ACM Trans. Inf. Syst. Secur.*, vol. 5, no. 3, pp. 203–237, 2002.
[11] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. of the 2001 IEEE Symp. on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 144–155.
[12] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proc. of the 2001 IEEE Symp. on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2001, p. 156.
[13] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, "Environment-sensitive intrusion detection." in *Proc. of the Recent Advances in Intrusion Detection (RAID)*, 2005, pp. 185–206.
[14] D.-Y. Yeung and Y. Ding, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, pp. 229–243, Jan. 2003.
[15] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," in *Proc. of the 2003 European Symp. on Research in Computer Security*, Gjøvik, Norway, Oct. 2003.
[16] G. Tandon and P. Chan, "Learning rules from system call arguments and sequences for anomaly detection," in *ICDM Workshop on Data Mining for Computer Security (DMSEC)*, 2003, pp. 20–29.
[17] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *IEEE Symp. on Security and Privacy*, 21-24 May 2006.
[18] R. G. Bace, *Intrusion detection*. Indianapolis, IN, USA: Macmillan Publishing Co., Inc., 2000.
[19] D. E. Denning, "An intrusion-detection model," *IEEE Trans. on Software Eng.*, vol. SE-13, no. 2, pp. 222–232, Feb. 1987.
[20] M. Burgess, H. Haugerud, S. Straumsnes, and T. Reitan, "Measuring system normality," *ACM Trans. Comput. Syst.*, vol. 20, no. 2, pp. 125–160, 2002.
[21] N. Ye and Q. Chen, "An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems," *Quality and Reliability Eng. Int'l*, vol. 17, no. 2, pp. 105–112, 2001.
[22] H. Debar, M. Becker, and D. Siboni, "A neural network component for an intrusion detection system," in *Proc. IEEE Symp. on Research in Computer Security and Privacy*, 1992.
[23] M. Theus and M. Schonlau, "Intrusion detection based on structural zeroes," *Statistical Computing & Graphics Newsletter*, vol. 9, pp. 12–17, 1998.
[24] A. K. Gosh, J. Wanken, and F. Charron, "Detecting anomalous and unknown intrusions against programs," in *ACSAC '98: Proc. of the 14th Ann. Computer Security Applications Conf.* Washington, DC, USA: IEEE Computer Society, 1998, p. 259.
[25] J. C. Galeano, A. Veloza-Suan, and F. A. Gonz?lez, "A comparative analysis of artificial immune network models," in *GECCO '05: Proc. of the 2005 Conf. on Genetic and evolutionary computation*. New York, NY, USA: ACM Press, 2005, pp. 361–368.
[26] S. Forrest, S. A. Hofmeyr, and A. Somayaji, "Computer immunology," *Commun. ACM*, vol. 40, no. 10, pp. 88–96, 1997.
[27] C. Ko, G. Fink, and K. Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," in *Proc. of the 10th Ann. Computer Security Applications Conf.*, vol. XIII. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994, pp. 134–144.
[28] W. Lee, S. J. Stolfo, and P. K. Chan, "Learning patterns from unix process execution traces for intrusion detection," in *Proceedings of the AAAI97 workshop on AI Approaches to Fraud Detection and Risk Management*. AAAI Press, 1997, pp. 50–56. [Online]. Available: citeseer.ist.psu.edu/lee97learning.html
[29] W. Lee and W. Fan, "Mining system audit data: opportunities and challenges," *ACM SIGMOD Rec.*, vol. 30, no. 4, pp. 35–44, 2001.
[30] C. Warrender, S. Forrest, and B. A. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *IEEE Symp. on Security and Privacy*, 1999, pp. 133–145.
[31] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrimination in a computer," in *SP '94: Proc. of the 1994 IEEE Symp. on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1994, p. 202.
[32] S. Zanero, "Behavioral intrusion detection," in *Proc. of the 19th Int'l Symp. on Computer and Information Sciences (ISCIS)*. Kemer-Antalya, Turkey: Springer-Verlag, Oct. 2004, pp. 657–666.
[33] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proc. of the 9th ACM Conf. on Computer and communications security*. New York, NY, USA: ACM Press, 2002, pp. 255–264.
[34] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 61–93, 2006.
[35] A. Stolcke and S. Omohundro, "Hidden Markov Model induction by bayesian model merging," in *Advances in Neural Information Processing Systems*, vol. 5. Morgan Kaufmann, 1993, pp. 11–18.
[36] A. Stolcke and S. M. Omohundro, "Inducing probabilistic grammars by bayesian model merging," in *Proc. of the 2nd Int'l Colloquium on Grammatical Inference and Applications*. London, UK: Springer-Verlag, 1994, pp. 106–118.
[37] S. Y. Lee, W. L. Low, and P. Y. Wong, "Learning fingerprints for a database intrusion detection system," in *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*. London, UK: Springer-Verlag, 2002, pp. 264–280.
[38] LibAnomaly. [Online]. Available: http://www.cs.ucsb.edu/~rsg/libAnomaly
[39] S. Zanero, "Unsupervised learning algorithms for intrusion detection," Ph.D. dissertation, Politecnico di Milano T.U., Milano, Italy, May 2006.
[40] G. H. Golub and C. F. V. Loan, *Matrix computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.
[41] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," in *Proc. of the IEEE*, vol. 77, 1989, pp. 257–286.
[42] W. R. Pestman, *Mathematical Statistics: An Introduction*. Walter de Gruyter, 1998.
[43] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "The 1999 DARPA off-line intrusion detection evaluation," *Comput. Networks*, vol. 34, no. 4, pp. 579–595, 2000.
[44] J. McHugh, "Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by lincoln laboratory," *ACM Trans. on Information and System Security*, vol. 3, no. 4, pp. 262–294, 2000.
[45] M. V. Mahoney and P. K. Chan, "An analysis of the 1999 DARPA / Lincoln laboratory evaluation data for network anomaly detection," in *Proc. of the 6th Int'l Symp. on Recent Advances in Intrusion Detection (RAID 2003)*, Pittsburgh, PA, USA, Sept. 2003, pp. 220–237.
[46] Shmoo Group. Capture the CTF. [Online]. Available: http://cctf.shmoo.com
[47] R. N. M. Watson and W. Salamon, "The FreeBSD audit system," in *UKUUG LISA Conf.*, Durham, UK, Mar. 2006.

Federico Maggi received a "laurea specialistica" (M.Sc.) degree in Computer Enginering from the Politecnico di Milano in December 2006; he is currently a PhD candidate in the same university and Visiting Research Scholar at the University of California Santa Barbara. His current research interests includes the security of computer systems and statistical data analysis. He is a member of the ISSA and of the IEEE.



Matteo Matteucci received a "laurea" (M.Sc.) degree from the Politecnico di Milano in 1999, a Master of Science in Knowledge Discovery and Data Mining at Carnegie Mellon University in 2002, and a Ph.D. in Computer Engineering and Automation from the Politecnico di Milano in 2003, where he is an Assistant Professor ("Ricercatore"). His current research interests are focused on Robotics and Machine Learning, mainly applying, in a practical way, techniques for adaptation and learning to autonomous systems. His research is in adaptive color models, robust tracking for video surveillance, reactive robot control, behavior modeling, intrusion detection, autonomous robots and learning machines in (i.e., neural network, decision trees, mixture models, etc.). He is a member of the IEEE.

Stefano Zanero received a "laurea" (M.Sc.) degree in Computer Engineering in December 2002, and subsequently in May 2006 a Ph.D. in Computer Engineering from the Politecnico di Milano, where he is currently an Assistant Professor ("Ricercatore"). His research interests include intrusion detection, web application security, and computer virology. He is a board member of the "Journal in Computer Virology", and has served as a reviewer for many primary international journals and conferences. He is a member of the IEEE, the IEEE Computer Society (for which he has been elected 2008 vice-chair of the Italy chapter), the ACM and of the ISSA (for which he serves since 2005 in the board of the Italy chapter, and since 2008 in the international Board of Directors).