Anomaly Detection with Diagnosis in Diversified Systems using Information Flow Graphs

Frédéric Majorczyk, Eric Totel, Ludovic Mé, Ayda Saïdane

Key words: anomaly detection, design diversity, COTS diversity, anomaly diagnosis, graph similarity

Abstract Design diversity is a well-known method to ensure fault tolerance. Such a method has also been applied successfully in various projects to provide intrusion detection and tolerance. Two types of approaches have been investigated: the comparison of the outputs of the diversified services without any knowledge of the internals of the server (black box approach) or an intrusive observation of the activities that occur on the diversified servers (gray box approach). Previous work on black-box approaches have shown that some types of attacks cannot be detected. In this paper, we introduce a gray-box approach, on the one hand to increase the detection coverage, and on the other hand to add some diagnosis capability to the IDS. Our gray-box approach is based on the comparison of information flow graphs generated by the activities on the servers.

1 Introduction

Intrusion detection includes misuse detection and anomaly detection. Misuse detection consists in detecting known attacks (and thus requires a base of signatures), as anomaly detection relies on the comparison of a system or application behavior with a previously defined "normal" behavior. Most of the time, anomaly detection requires to explicitly build the model of the normal behavior, either statically or dynamically (e.g., during a learning phase). Previous work [9, 6, 8] has introduced a

Frédéric Majorczyk, Eric Totel, Ludovic Mé
Supelec, Avenue de la Boulaie, 35575 Cesson-Sévigné Cedex, France, e-mail: frederic.majorczyk@supelec.fr, eric.totel@supelec.fr, ludovic.me@supelec.fr

Ayda Saïdane
University of Trento, via Belenzani, 12 I-38100 Trento, e-mail: ayda.saidane@unitn.it

way to avoid building the behavior model explicitly, while allowing the built IDS to detect new or unknown attacks. This previous work is based on a dependability technique: N-version programming [1]. However, instead of developing specifically each variant like in classical N-version programming, latest work proposes the use of COTS (Components Off The Shelf) components. This reduces the cost of the architecture, and thus appears as the only viable approach, from an economic point of view. Nevertheless, the detection relies on the same hypothesises as in classical N-Version programming: the faults in the COTS (and thus the intrusions) are decorelated, to ensure that an intrusion on one of the server cannot occur on the others.

The basic approach of the intrusion detection consists in comparing the outputs of the diversified servers. Two types of comparisons can be performed at the IDS level as the servers can be considered as black-boxes or gray-boxes. In the first case, the outputs considered are the outputs of the servers. In the second case, the outputs are composed of both the internals of the servers (e.g., system calls) and the outputs of the servers.

In this paper, our objective is to present a new IDS based on COTS diversity, on one hand to correctly handle all types of attacks, on the other hand to add diagnosis capabilities to the intrusion detection system. We propose here an approach that exploits information about the activities occurring in the diversified servers: our gray box approach consists in dynamically building a view of the information flows that occur in the system, in order to be able to compare the behaviors of several diversified servers. The detection relies on the creation and comparison of information flow graphs generated by the activities on the servers.

The method we introduce provides the security administrator with an insight of what happens in the different servers. This brings some diagnosis capabilities to an anomaly detector. Indeed, a major drawback of anomaly detection is that no evidence is provided about the cause of the anomaly. No diagnosis is performed in order to help the administrator to identify whether an alert is a false positive or not; in the case of a true positive, no information is provided about the intrusion that has led to the anomaly.

In the following sections, we consider the various approaches that have been carried out in the domain of implicit model based anomaly detection (Section 2), and propose a technique to detect behavior variations of the diversified COTS in the architecture (Section 3). Then we show how we can propose a diagnosis of an intrusion using the graphs (Section 3.4). Finally a prototype is described to demonstrate the feasibility of the approach (Section 4).

2 Related Work

We present here the different work that have been carried out in the context of black-box and gray-box approaches.

Black-Box Intrusion Detection using Diversity

Three recent projects use diversity to detect intrusions with a black-box approach: DIT, HACQIT and DADDi.

DIT (Dependable Intrusion Tolerance) [9] is a project that proposes a general architecture for intrusion-tolerant systems and the implementation of an intrusion-tolerant web server as a specific instance. The architecture includes functionally redundant COTS servers running on diversified operating systems and platforms, hardened intrusion-tolerant proxies that mediate client requests and verify the behavior of servers and other proxies, and monitoring and alert management components based on the EMERALD intrusion-detection framework [7]. The architecture was then extended to consider the dynamic content issue and the problems related to on-line updating. The comparison of outputs is based on MD5 hashes of the web pages but the intrusion detection relies mainly on the host monitors and network intrusion detection systems.

HACQIT [6] (Hierarchical Adaptive Control for QoS Intrusion Tolerance) is a project that aims at providing intrusion tolerance for web servers. The architecture is made up of two COTS web servers: an IIS server running on Windows and an Apache server running on Linux. One of the servers is declared as the primary and the other one as the backup server. Only the primary server is connected to users. Another computer, the Out-Of-Band (OOB) computer, is in charge of forwarding the request of each client from the primary server to the backup one, and of receiving the responses from each server. Then, they compare the responses given by each server. The comparison is based on the status code of the HTTP response. In addition to this detection mechanism, host monitors, application monitors, a network intrusion detection system (Snort) and an integrity tool (Tripwire) are also used to detect intrusions.

DADDi [8] (Dependable Anomaly Detection with Diagnosis) implements an IDS for web servers with an architecture composed by three different COTS servers: an Apache on Mac-OS X, an IIS on Windows 2000 and a thttpd on Linux. The project extends the comparison to the complete network output of the COTS servers. Unlike the two projects presented above, the intrusion detection relies only on diversity. Neither mechanisms nor IDSes are used. The authors show that, depending on the algorithm used, the COTS diversity method can lead to many false positives due to design and specification differences. To solve this key issue, they propose the introduction of masking mechanisms to determine which output differences are the consequence of a design or a specification difference. This provides a model of the normal differences, which is much simpler to build than a complete web server model.

These three projects adopt a black-box approach. One major drawback is then that they are not able to detect all intrusions since they do not consider all outputs of the monitored services but only the network outputs. An intrusion against integrity can be missed: an attacker who has successfully compromised one server can forge a response identical to the ones from the other servers. It is necessary to add host and application monitors to be able to detect this kind of intrusions. Both the HACQIT and DIT projects add host monitors to detect this kind of intrusions but these host monitors do not use diversity. Gray-box approaches would also be able to detect this kind of intrusions since they monitor and compare the internal activity of the system.

Gray-Box Intrusion Detection using Diversity

Gao, Reiter and Song [4, 5] propose a way to compare system call sequences performed by different COTS on different operating systems. They introduce the notion of behavioral distance which is a measure of the deviation of the behaviors of two processes. They propose two ways to compute this distance: using evolutionary distance [4] and hidden Markov models [5]. The key idea is that an intrusion should thus modify the behavior of only one of the processes and should increase the behavioral distance. If the distance computed is above a given threshold then an alert is emitted. A drawback of this work is that it only takes the number of the system calls into account, while the operation performed by a system call often depends on its arguments. Moreover, no diagnosis of the alerts is provided to the administrator.

3 Intrusion Detection and Diagnosis by Comparison of Information Flow Graphs

Classical N-version programming requires to define which outputs must be compared, the system detects only errors that are propagating through these outputs. In the context of a black-box detection, only the intrusions affecting the server network outputs can be detected. In the context of a gray-box detection, other outputs, like system calls, can be compared. This approach is the one that has been used by [4, 5]. However, it is not straightforward to compare system calls on different operating systems as they are not equivalent namely and functionally.

The different server versions should behave the same with respect to the security policy. This security policy is generally implemented using access rights, and can be described as a set of permitted information flows in the system. This implies that a faulty service will not only invoke unauthorized system calls, but also produce illegal information flows generated by system calls. For example, an intrusion against confidentiality is seen as an illegal information flow between two objects. All the information flows generated by the processing of a request form an information flow graph. We argue here that comparing two information flow graphs, while not trivial,

is easier than comparing two sequences of system calls produced on two different systems. An other advantage is that we do not have to monitor all the system calls, but only the subset of them that generates information flows.

In the following Sections we describe what is an information flow graph (Section 3.1), and the method we propose to use in order to compute the similarity between graphs (Section 3.2). Then we apply this method to detect intrusions in an architecture of diversified COTS servers (Section 3.3).

3.1 Information Flow Graphs

First of all, we must define the information flow notion. We consider an information flow has been produced from an object o_1 to an object o_2 if the state of the object o_2 causally depends [3] on the state of the object o_1.

An information flow graph is a set of information flows and objects that are involved during an invocation of the diversified service.

Formally, an information flow graph is a labeled graph, i.e., a directed graph $G = (V, r_V, r_E)$, of information flows between objects in the operating system, where: V is a set of vertices, $r_V \subseteq V \times L_V$ is the relation between the vertices and the vertex labels (L_V is the set of vertex labels), $r_E \subseteq V \times V \times L_E$ is the relation between the edges and the edge labels (L_E is the set of edge labels).

The vertices correspond to objects of the operating system. Currently, we consider processes, threads, files, sockets, pipes and memory mappings. The edges correspond to information flows between these objects. The labels are needed to calculate the similarity between two graphs, as explained in the next section.

In practice, each vertex or edge is associated with data items that define information required to characterize the edge or the vertex. For example, vertices are associated with a type (Process, Socket, File, Pipe, Mapping), edges with one of the types [1] (Process_to_file, File_to_process, Process_to_socket, ...). Moreover, depending on this type, additional information can be attached to each edge or vertex. For example, the vertices of type File are associated with a name, the file descriptor, the creation time and the destruction time of the file descriptor in the OS. Processes are associated with their name, their pid, the pid of their parent and a creation and destruction time. The edges are associated with the data transferred between the source and the destination of the flows as well as the time of the call.

The labels are in fact defined as a part of these data items chosen to compute the similarity, as explained in the following section: in our prototype, we use only the type as label. This implies that in our prototype, the data are not taken into account when we compute the similarity. In fact, we check the existence of information flows, instead of the contents of the information flows. This choice has been made for performance reasons.

[1] This information can seem redundant with the types associated with the vertices, but are in fact required to compute the graph similarity, see Section 3.2.

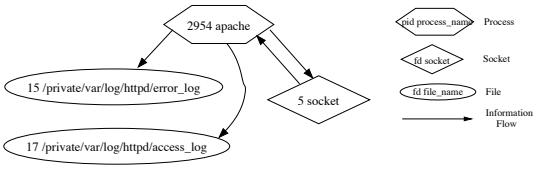

Fig. 1 Information flow graph for a HTTP request on the Apache web server

An example of information flow graph can be seen on Figure 1. This information flow graph is the one obtained for a single HTTP request. The type of a vertex is represented by its shape and information about vertices (name, pid, fd) is written inside. The graph Figure 1 shows that the Apache process reads from a socket, writes to two log files and writes to the socket (the time associated with the information flows allow to determine the chronology of the flows). The Section 4.2 explains how such a graph is built in practice, from system call monitoring.

3.2 Information Flow Graph Similarity

Since the COTS servers implement the same service, we expect that the information flow graphs on the different servers are quite similar. We need a way to assess if two graphs are similar or not. We use the model developed by Champin and Solnon [2] to measure the similarity between two labeled graphs. In their work, they propose an algorithm to calculate this similarity that we slightly change to optimize the computation to our particular case. We briefly present their approach and then detail the algorithm we use to compare information flow graphs.

To define the similarity between two graphs, we need to define the descriptor of a labeled graph: the descriptor of a labeled graph $G = (V, r_V, r_E)$ is defined by $desc(G) = r_V \cup r_E$.

We consider two labeled graphs $G_1 = (V_1, r_{V_1}, r_{E_1})$ and $G_2 = (V_2, r_{V_2}, r_{E_2})$ and we look for a measure of the similarity between those two graphs. In order to measure this similarity, the notion of mapping is defined: a mapping m is a relation $m \subseteq V_1 \times V_2$. m is a set of pairs of vertices. It must be noted that, in a mapping m, a vertex $v_1 \in V_1$ (resp. $v_2 \in V_2$) can be mapped with zero, one or more vertices in V_2 (resp. V_1). A functional notation may be used for m: $m(v)$ is the set of vertices which the vertex v is mapped with.

The similarity between G_1 and G_2 with respect to a mapping m is given by:

$$sim_m(G_1, G_2) = \frac{f(desc(G_1) \sqcap_m desc(G_2))}{f(desc(G_1) \cup desc(G_2))}$$

where f is a non-decreasing positive function with respect to inclusion (the cardinality is a function that respects these criteria, for example) and $\sqcap_m$, the intersection

with respect to a mapping represents the set of labels corresponding to the vertices and to the edges in the mapping m:

$$\begin{aligned} desc(G_1) \sqcap_m desc(G_2) = & \{(v,l) \in r_{V_1} | \exists v' \in m(v), (v',l) \in r_{V_2}\} \\ & \cup \{(v,l) \in r_{V_2} | \exists v' \in m(v), (v',l) \in r_{V_1}\} \\ & \cup \{(v_i,v_j,l) \in r_{E_1} | \exists v'_i \in m(v_i), \exists v'_j \in m(v_j)(v'_i,v'_j,l) \in r_{E_2}\} \\ & \cup \{(v_i,v_j,l) \in r_{E_2} | \exists v'_i \in m(v_i), \exists v'_j \in m(v_j)(v'_i,v'_j,l) \in r_{E_1}\} \end{aligned}$$

A last concept is necessary to measure the similarity between two graphs. A vertex can indeed be mapped with more than one vertex. It can be interesting in our case, as we may need to map two processes in one operating system with one process in another operating system. The notion of *splits* can be added to take into account the mapping of a particular vertex to multiple vertexes. The *splits* of a mapping m represents the vertices that are mapped with more than one vertex in the mapping m:

$$splits(m) = \{(v,s_v) | v \in V_1 \cup V_2, s_v = m(v), |m(v)| \geq 2\}$$

The definition of the similarity with respect to a mapping m is changed to:

$$sim_m(G_1,G_2) = \frac{f(desc(G_1) \sqcap_m desc(G_2)) - g(splits(m))}{f(desc(G_1) \cup desc(G_2))}$$

where g is a positive, monotonic and non-decreasing function with respect to inclusion. We have defined the similarity between two graphs with respect to a mapping m. The similarity between two graphs can be defined by:

$$sim(G_1,G_2) = \max_{m \subseteq V_1 \times V_2} \frac{f(desc(G_1) \sqcap_m desc(G_2)) - g(splits(m))}{f(desc(G_1) \cup desc(G_2))}$$

So finding the similarity between two graphs G_1 and G_2 means finding the mapping m that maximizes this value.

Champin and Solnon [2] propose two algorithms to solve this problem: a complete search with some optimizations that allows to cut branches off the search tree and a greedy algorithm. In our prototype, we have used the complete search algorithm as we are more interested in the detection accuracy than in the temporal performance of the similarity computation, at least for a first implementation.

3.3 Intrusion Detection using Graph Similarity

In information flow graphs, an intrusion is characterized by the creation or modification of information flows, active objects (e.g., processes) and/or passive objects (e.g., files). An intrusion against confidentiality implies the creation of information flows and if necessary, the creation of new objects. An intrusion against integrity is

characterized by the creation or modification of information flows and if necessary, the creation of new objects. Thus, an intrusion affects the value of the similarity between the information flow graph of a successfully attacked server and the one of a server which has not been compromised.

Similarity Threshold. The calculation of a similarity is a function taking two graphs as parameters. A high similarity means that the servers have behaved quite in the same way. A low similarity means that the servers have behaved quite differently, which can be the consequence of an intrusion, a design difference or specification difference. Our approach consists in learning what is an acceptable value of a similarity in the context of a normal behavior. This leads to determine a threshold to decide when an alert must be emitted, i.e., to determine the value of the similarity, under which it is symptomatic of an intrusion. If a similarity is lower than the threshold, we generate an alert.

This threshold must be determined experimentally for each pair of diversified services: the similarity depends on the cardinality of the descriptors of the graphs considered and then depends on the application considered. If the graphs are large, one or more vertices or edges that are not mapped have less influence on the similarity than in small graphs. So there cannot exist a unique threshold for all the applications. In order to calculate this threshold, we determine statistically which value corresponds to a normal behavior, i.e., a request that is not an attack.

Intrusion Detection Algorithm. In the context of an architecture with n COTS servers S_i, we must determine the similarity threshold $t_{i,j}$ for each pair of servers as explained in the previous paragraph. Detecting an intrusion requires calculating the similarities between all pairs of servers for a given service request. This leads to the computation of $C_n^2 = \frac{n!}{(n-2)!\times 2!}$ graph similarities, noted $s_{i,j}$. Since the similarity is symmetric, we can write $s_{i,j} = s_{j,i}$ for all (i,j) in $\{1,n\}^2$. We note $I_1^{i,j} = [0, t_{i,j}]$ and $I_2^{i,j} = [t_{i,j}, 1]$.

Currently, we use the following rules to determine the decision of our gray-box IDS for n servers:

$$\exists (i,j) \in \{1,n\}^2, i < j, s_{i,j} \in I_1^{i,j} \Rightarrow Alert$$
$$\forall (i,j) \in \{1,n\}^2, i < j, s_{i,j} \in I_2^{i,j} \Rightarrow No\ Alert$$

i.e., an alert is emitted as soon as one similarity is low, i.e., beneath $t_{i,j}$. If all the similarities are high, i.e., above $t_{i,j}$, no alert is emitted.

Intrusion Localization. Low similarities indicate that an incorrect activity has occurred in the architecture. Nevertheless, under the hypothesis that only one server can be compromised at a time, this server must be the only one leading to low similarities. In that case, the localization of the compromised server is possible. This localization is required to apply a reconfiguration to the architecture in order to mask the effects of the detected intrusion (e.g., the reconfiguration of a server). In

	$s_{2,3} \in I_1^{2,3}$		$s_{2,3} \in I_2^{2,3}$	
$s_{1,3} \in$ / $s_{1,2} \in$	$I_1^{1,3}$	$I_2^{1,3}$	$I_1^{1,3}$	$I_2^{1,3}$
$I_1^{1,2}$	$A/?$	A/S_2	A/S_1	$A/?$
$I_2^{1,2}$	A/S_3	$A/?$	$A/?$	NA

Table 1 Alerts and localization of the server compromised in the case $N = 3$; A means Alert (gray cells), NA means No Alert (white cells), ? means no localization is possible, S_i means the server S_i is considered as being compromised

our prototype, the localization of the server compromised is based on the following rule:

$$\left.\begin{array}{l} \exists i \in \{1,n\}, \forall j \in \{1,n\}, j \neq i, s_{i,j} \in I_1^{i,j} \\ \wedge \\ \forall (k,l) \in \{1,n\}^2, k \neq i, l \neq i, s_{k,l} \in I_2^{k,l} \end{array}\right\} \Rightarrow S_i \text{ is compromised}$$

Three Server Instance. Table 1 sums up, in the case of three servers S_1, S_2 and S_3, in what conditions we decide to raise an alert and if we can localize the compromised server in function of the computed similarities.

Some cases should not happen: for example, for three servers, if $s_{1,2}$ and $s_{2,3}$ are high and $s_{1,3}$ is low. This means that the behaviors of S_1 and S_2 are close as well as the ones of S_2 and S_3, but the behaviors of S_1 and S_3 are really different. Since we have no evidence about the transitivity of the relation linked to the similarity, we consider that this case may be possible. An alert is emitted by the IDS but no localization is possible.

3.4 Diagnosis of Anomalies Detected

In classical anomaly detector, no diagnosis is associated with the alerts, which is one of the main drawbacks of this kind of detector. Here, as we capture information flows that can be viewed as an history of the system activity, we provide the security administrator with an evidence of what happens in the different servers: it is possible to explain an intrusion through the differences between the graphs: processes created, files read or written, sockets opened, etc.

By computing the similarity between the graphs, we identify also the active objects (processes), the passive objects (files, sockets, pipes, ...) and the information flows not mapped in the best mapping, i.e., the one for which the similarity is maximum. It must be noted that the best mapping depends on the computation of the similarity and thus on the functions f and g.

In case of an intrusion, the objects not mapped are visible effects at the OS level of an intrusion. By analyzing these objects, it is possible to gather some information about the intrusion: processes created, files written or read, sockets created, etc. If it is not possible to identify directly the vulnerability exploited, this information may

lead to it. In the case of a zero-day, it even offers a good starting point to discover the currently unknown vulnerability.

We propose to show to the security operator the graphs of the different servers for the suspicious input and mark the objects and flows not mapped (This is illustrated in Section 4.3.4). We believe that this approach can be very helpful to a security operator and can be extended by automatically summing up the objects non mapped and their interactions. It is, as far as we know, the first anomaly-based approach in intrusion detection, which offers such a diagnosis capability.

4 Prototype and Experimental Results

We have implemented a proof-of-concept prototype of an IDS based on information flow graph similarity for web servers. After a brief presentation of the components of the architecture, we discuss the modeling of system calls. Finally some results show the performances of this prototype, in relation to its detection capabilities.

4.1 Intrusion Detection Architecture

We use three different web servers in our prototype: a thttpd (for tests with a static web server) or Lighttpd (for tests with a dynamic web server) web server running on Linux, an Abyss web server running on Windows 2000 Server and an Apache web server running on Mac-OS X. The architecture (Figure 2) is composed of several components: a HTTP proxy, a gray-box IDS based on information flow graph comparison and, on each server, a wrapper, a graph generator and a system call logger. The role of the wrapper is mainly to associate a request with its beginning and ending times. It receives an HTTP request from the proxy, stores the beginning time of the request and forwards the request to the web server. Then the wrapper forwards the response of the web server to the proxy and stores the time at the end of the response. Then it asks the graph generator for the information flow graph corresponding to the request by sending the beginning time and the ending time of the request considered and send the information flow graph to the proxy. Depending on the design of the web servers, it is not always easy to determine the correspondence between system calls and requests. To solve this problem, we choose in our prototype to serialize the requests to ensure that at one time only one request is processed. This proof-of-concept prototype has been developed to evaluate detection precision and reliability, its optimization in terms of real-time capability is left for future work.

The system call logger logs the system calls performed by the web server and sends them to the graph generator on demand. The graph generator builds information flow graphs from system calls and sends them to the wrapper.

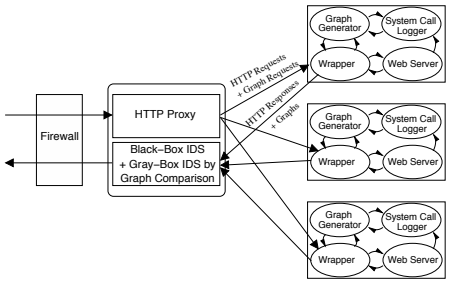

Fig. 2 Gray-box intrusion detection architecture

4.2 Monitoring System Calls Generating an Information Flow

About twenty system calls have been identified to generate information flows (For example, the *read* system call) and are thus monitored on each operating system. In the case of the *read* system call, the graph built is the following: the process which performs the system call *read* is an active object, the file, socket or pipe read is a passive object, and these two objects are linked.

However, some other information flows are difficult to build. The system calls that create another thread such as *clone* (on Linux) allow information flows between processes. In case of the system call *clone*, the child process is a copy of his parent process. The child process and the parent process share their memory and can thus communicate. It is possible to monitor whether one of the processes access to this data, but this has a significant impact on the performance. So we decided to model this system call by two information flows: one from the child to the parent process and the other one from the parent to the child. The information flow created this way has an empty label data.

A system call that creates another process such as *fork* corresponds to the execute operation of our model. For the same reason as the one mentioned above, the information flow which corresponds has an empty label data.

System calls such as *mmap* can also be the source of information flows which can not be modeled without observing memory accesses of processes. *mmap* allows a process to map a file in memory. Reading or writing in the file is simply performed by reading or writing to memory. Depending on the arguments of *mmap*, we decided to create information flows between the process which executes *mmap* and the file considered. If the mapping is read-only (resp. write-only), we create an information flow from the file (resp. process) object to the process (resp. file) object. If the mapping is read-write, we create two information flows: one from the file to the process object and the other one in the other way.

Our current prototype does not consider some IPC mechanisms (messages, shared memory) and signals. IPC mechanisms can be modeled as passive objects. Signals create information flows between the process that sends the signal and the

process that receives it. They can be modeled by an information flow with a label data corresponding to the number of the signal sent.

4.3 Experimental Results

4.3.1 Detecting a Successful Attack Against Integrity

One of the motivations of defining a gray-box approach is to be able to detect successful attacks against integrity, where the network answers of the diversified servers does not reflect the intrusion. To demonstrate this detection capability, we have developped a small diversified php script, where, on the value of one parameter, a file write operation is performed on one server, but all the scripts return the same answer. This attack results in the creation of an information flow graph containing a write operation on the attacked server, and thus results in a low similarity between this server and the others. Consequently, this approach complementarize a black-box approach, where this type of attack were impossible to detect.

4.3.2 Evaluation of the False Positive Rate

In this Section, we evaluate the false positive rate of our prototype. For that purpose, we use a static web server, the one of our campus, and two sets of normal requests (real requests to this server).

Computation of the Thresholds $t_{i,j}$. As stated in Subsection 3.3, the thresholds $t_{i,j}$ must be chosen experimentally for each pair of servers used. We decide to set up the threshold so as to ensure that, for most of normal HTTP requests, our prototype does not raise an alert, i.e., the similarities computed for these requests are above the thresholds $t_{i,j}$.

For the normal requests, we use a set of HTTP requests logged on the website of our campus during a week. This set is composed of 71,596 HTTP requests. To check if this set contains only non-intrusive requests, we use WebSTAT [10] and the black-box IDS of [8]: all alerts generated have been confirmed to be false positives. Thus we have a high confidence in the fact that the traffic does not contain successful attacks. After this phase, we decide to set up all the thresholds to 0.7. More than 99.5% of the requests are thus considered as normal.

Results. In order to evaluate the false positive rate, we use another set of HTTP requests logged by the server of our campus during a week. This set is composed of 105,228 requests.

Table 2 sums up the 140 alerts raised by the gray-box IDS and the localization of the server considered as being compromised. All these alerts are false positives since they are not due to intrusions. It represents a false positive rate of 0.13% and 20 alerts a day, which is acceptable.

	$s_{2,3} \in I_1^{2,3}$		$s_{2,3} \in I_2^{2,3}$	
$s_{1,2} \in$ \ $s_{1,3} \in$	$I_1^{1,3}$	$I_2^{1,3}$	$I_1^{1,3}$	$I_2^{1,3}$
$I_1^{1,2}$	0 (?)	0 (S_2)	120 (S_1)	1 (?)
$I_2^{1,2}$	2 (S_3)	1 (?)	16 (?)	105,088

Table 2 Number of alerts and localization of the server considered compromised for the test week; gray cells mean an alert is raised, white cells mean no alert is raised, ? means no localization is possible, S_i means the server S_i is considered compromised

Similarity between \ Intrusions	SQL injection	write	execute 'whoami'	XSS (insertion)	XSS (read the corrupted entry)
Lighttpd and Apache	0.9655	0.725	0.5882	0.9677	0.9143
Apache and Abyss	0.7742	0.9715	0.6364	0.9677	0.9706
Lighttpd and Abyss	0.7838	0.7209	0.6667	1	0.9189
	detected	detected	detected	not detected	not detected

Table 3 Similarities between the graphs of the different servers for the intrusive requests

4.3.3 Detection Capabilities

In this Section, we test our prototype in the context of a dynamic web server and evaluate its detection capabilities.

We replace the thttpd server by the Lighttpd server on the Linux machine. For the web site, we choose an application named Bibtex Manager which is written in php and uses a database as a backend. This application manages a database of Bibtex citations. We log 86 HTTP requests that represent a normal use of the application. We introduce four vulnerabilities in one of the versions and develop corresponding exploits.

The similarities between the different graphs corresponding to the intrusions are lower than the ones for normal requests. While the similarity for normal requests is, in the mean, around 0.95, the similarity between the graph of a compromised server and a server not compromised is lower than 0.8. Table 3 sums up the similarities obtained for the intrusions.

By setting the threshold for each pair of servers to 0.8, our prototype IDS is able to detect all the intrusions except the XSS attack.The XSS attack is not detected as its impact on the similarities (see Table 3) is too low.

4.3.4 Diagnosis Capabilities

In order to show the diagnosis capabilities of our approach, we performed an intrusion against a tiny vulnerable web server developed for educational purpose (the result appears on Figure 3). The intrusion consists in a directory traversal so as to

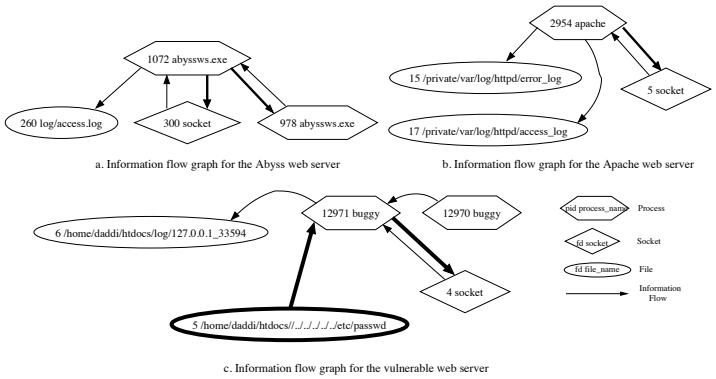

Fig. 3 Three information flow graphs linked to a HTTP request on different servers and identification of the objects not mapped

read the file */etc/passwd*. This server does not check the presence of '../' string in the url while the other servers refuse to serve such a request.

The file object representing the *passwd* file is not mapped with any objects in the other graphs as well as the information flows representing the read access to the *passwd* file and the write access to the socket. These objects are emphasized on Figure 3. For the Apache web server as well as for the Abyss web server, the information flows representing the write access to the socket are not mapped with their equivalent in the graph for the vulnerable server but are associated with each other.

5 Conclusion

Detecting intrusions by using COTS diversity proved to provide interesting results, both in terms of false positives and false negatives. However, the black-box approaches that have been investigated suffer from being incomplete, as they do not permit to detect intrusions that have no impact on the network outputs. To improve them, we have thus developed a gray-box approach that implements information flow graph comparisons.

As many IDSes, the method proposed can miss some attacks, if the objects involved in the system do not produce sufficient differences in the information flow graphs built on the different servers. This can lead to some false negatives. However, further work on the definition of the comparison algorithm (and the function f and g) should reduce the false negative rate.

Moreover, the current prototype is still far from being perfect, as it generates a too high false positive rate and exhibits low time performances. These flaws can be corrected by introducing design difference masking mechanisms on one hand

(similarly to the black box approach in [8]), and a greedy algorithm on the other hand to decrease the similarity computation time significantly. This will motivate our future work.

Despite these current limitations, this approach and the experiments carried out with our proof-of-concept prototype show that our gray-box IDS is capable of detecting intrusions of all kinds as they imply a difference in the information flows observed in the servers.

Finally, the analysis of the differences between the information flow graphs has proved to be efficient to bring diagnosis capabilities to the IDS as it enlighten the effects of the intrusions at the OS level. The advantage of our approach is thus to propose to the administrator more than a simple intrusion detection mechanism: it brings him an evidence of the intrusion and its causes.

Acknowledgements This work has been funded by the region Bretagne and the French National Research Agency in the context of the DADDi project.

References

1. Bharathi, V.: N-version programming method of software fault tolerance: A critical review. In: National Conference on Nonlinear Systems and Dynamics (NCNSD). Kharagpur, India (2003)
2. Champin, P.A., Solnon, C.: Measuring the similarity of labeled graphs. In: in Proceedings of the 5th International Conference on Case-Based Reasoning (ICCBR 2003), pp. 80–95. Trondheim, Norway (2003)
3. d'Ausbourg, B.: Implementing secure dependencies over a network by designing a distributed security subsystem. In: Proceedings of the European Sysmposium on Research in Computer Security (ESORICS'94), pp. 249–266 (1994)
4. Gao, D., Reiter, M.K., Song, D.: Behavioral distance for intrusion detection. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), pp. 63–81. Seattle, WA (2005)
5. Gao, D., Reiter, M.K., Song, D.: Behavioral distance measurement using hidden markov models. In: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006), pp. 19–40. Hamburg, Germany (2006)
6. Just, J.E., Reynolds, J.C., Clough, L.A., Danforth, M., Levitt, K.N., Maglich, R., Rowe, J.: Learning unknown attacks - a start. In: A. Wespi, G. Vigna, L. Deri (eds.) Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), *Lecture Notes in Computer Science*, vol. 2516, pp. 158–176. Zurich, Switzerland (2002)
7. Porras, P.A., Neumann, P.G.: EMERALD: Event monitoring enabling responses to anomalous live disturbances. In: Proc. of the 20th National Information Systems Security Conference, pp. 353–365. Baltimore, MD (1997). URL http://www2.csl.sri.com/emerald/emerald-niss97.html
8. Totel, E., Majorczyk, F., Mé, L.: COTS diversity based intrusion detection and application to web servers. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), pp. 43–62. Seattle, WA (2005)
9. Veríssimo, P.E., Neves, N.F., Correia, M.P.: Intrusion-tolerant architectures: Concepts and design. In: Architecting Dependable Systems, *Lecture Notes in Computer Science*, vol. 2677. Sptringer-Verlag (2003)
10. Vigna, G., Robertson, W., Kher, V., Kemmerer, R.A.: A stateful intrusion detection system for world-wide web servers. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003), pp. 34–43. Las Vegas, Nevada (2003)