UNIVERSITÀ DEGLI STUDI DI TRIESTE
Sede Amministrativa del Dottorato di Ricerca

XX CICLO DEL
DOTTORATO DI RICERCA IN
INGEGNERIA DELL'INFORMAZIONE

Techniques for Large-Scale Automatic Detection of Web Site Defacements

(Settore scientifico-disciplinare ING-INF/05)

DOTTORANDO
Eric Medvet

COORDINATORE DEL COLLEGIO DEI DOCENTI
Chiar.mo Prof. **Alberto Bartoli**
Università degli Studi di Trieste

RELATORE
Chiar.mo Prof. **Alberto Bartoli**
Università degli Studi di Trieste

UNIVERSITÀ DEGLI STUDI DI TRIESTE
Sede Amministrativa del Dottorato di Ricerca

XX CICLO DEL
DOTTORATO DI RICERCA IN
INGEGNERIA DELL'INFORMAZIONE

Techniques for Large-Scale Automatic Detection of Web Site Defacements

(Settore scientifico-disciplinare ING-INF/05)

DOTTORANDO
Eric Medvet

COORDINATORE DEL COLLEGIO DEI DOCENTI
Chiar.mo Prof. **Alberto Bartoli**
Università degli Studi di Trieste

FIRMA: ....

RELATORE
Chiar.mo Prof. **Alberto Bartoli**
Università degli Studi di Trieste

FIRMA: ....

Contents

Chapter 1

Introduction

The web has become an essential component of our society. A huge number of organizations worldwide rely on the web for their daily operations, either completely or only in part. Nowadays, the confidence in an organization heavily depends on the quality of its web presence, which must convey a sense of trust and dependability to its users over the time. Any attack that affects the quality of such web presence may thus cause a serious damage to the organization, as well as to its users or customers. Unfortunately, incidents of this sort are common: more than 490,000 web pages were *defaced* in 2005, and the number of such attacks is constantly growing in the recent years [83].

One may try to prevent such kind of incidents, by applying a set of security policies; alternatively, one can try to detect whether such an attack has been successfully carried out. Being able to promptly detect a defacement is important, since it allows to mitigate it and hence to limit the damage that it causes to the organization and to its users or customers. Current statistics about web site defacement seem to confirm that this kind of attack does affect organizations exposed on the web; hence, simply relying on prevention might not be an effective approach to cope with this problem.

Most web sites lack a systematic surveillance of their integrity and the detection of web defacements is often demanded to occasional checks by administrators or to feedback from users. There exist technologies for *automatic* detection of web defacements that are meant to be run by the organization hosting the web site to be monitored, because such technologies require the installation of dedicated appliances within the organization (see Chapter 2 for a more complete analysis). Essentially, all such technologies are based on a comparison between the web page (or, more broadly, web resource) and an uncorrupted copy kept in a safe place. Whenever the web resource is modified, the *trusted copy* must be correspondingly updated, which usually requires some restructuring of the operational processes of the organization. While this approach may indeed be effective, in practice very few organizations actually deploy such technologies.

This thesis describes our research work which focused on designing, analyzing and testing a framework for automatic detection of web site defacements. It is aimed at proposing a solution that should be really feasible for a large number of organizations.

Two key requirements in this respect are the following. First, the tool should not require the organization to provide any baseline content nor to keep it updated—that is, it should not follow the trusted copy approach. Second, the tool should not require any installation on the organization site. For these reasons, it has been chosen to use an approach based on *anomaly detection*.

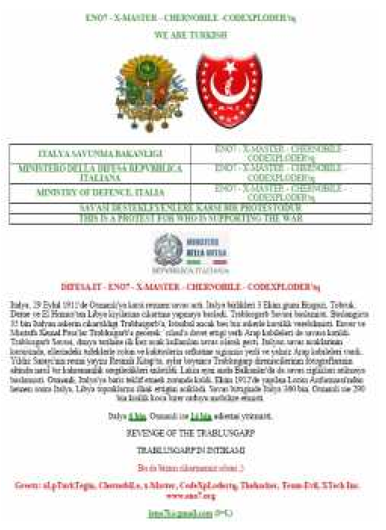
Anomaly detection is widely used in Intrusion Detection Systems (IDSs) [16]: it consists in generating a profile of the monitored system and then detect deviations from the profile, on the assumption that any deviation is a sign of an attack. The profile is generated automatically starting from a set of observations of the system. Anomaly detection is the counterpart of signature detection: the latter approach consists in looking for signs of known attacks. Signature detection is very effective on known attacks but it is unable to detect unknown attacks; on the other hand, it usually exhibits a very low number of false positives, in contrast to anomaly detection.

Anomaly detection approach has been used in this proposal as follows. A tool has been built that periodically checks the content and appearance of many remote web pages and compares each of them against their corresponding profiles. If the current content and appearance do not match with the profile, some kind of alert is generated. The key idea is quite simple; nevertheless, its actual implementation involves tackling several specific problems which were not solved previously. What kind of data should be extracted from a web resource? How should be built the profile with them? Is anomaly detection practically effective with highly dynamic web resources? The following chapters describe the results of our investigations about all these issues.

The main motivation for our work springs from this observation: in our surroundings there is a plenty of web sites which many people depend upon, yet, most of the organizations hosting these sites have neither the expertise nor the budget necessary to acquire, install, integrate and maintain one of the existing technologies for detecting web defacements automatically. In our administrative region alone there are literally hundreds of web sites that fall in this category and are representative of organizations of the civil administration (indeed, lack of adequate security budgets is quite a widespread problem today [31]). The potential effects of malicious intrusions on these sites can be imagined easily. We believe that the framework that we propose could help in changing this scenario. A monitoring service that may be joined with just a few mouse clicks and does not impose any burden on the normal operation of a web site, would have the potential to be really used widely. Such a service, thus, could greatly improve robustness and dependability of the web infrastructure.

1.1 Web site defacement: a brief overview

Web site defacements is a remote attack which consists in modifying a web page in an unauthorized way. It has become a fact of life in the Internet, similarly to phishing, worms, denial of service and other resembling phenomena. The significance of web site defacement is manifest: Zone-H, a public web-based archive devoted to gathering evidences of defacements (http://www.zone-h.org), collected approximately 490,000 de-



ENO7 - X-MASTER - CHERNOBILE - CODEXPLODER'tq

WE ARE TURKISH

ITALYA SAVUNMA BAKANLIGI	ENO7 - X-MASTER - CHERNOBILE - CODEXPLODER'tq
MINISTERO DELLA DIFESA REPUBBLICA ITALIANA	ENO7 - X-MASTER - CHERNOBILE - CODEXPLODER'tq
MINISTRY OF DEFENCE, ITALIA	ENO7 - X-MASTER - CHERNOBILE - CODEXPLODER'tq
SAVASI DESTEKLEYENLERE KARSI BIR PROTESTODUR	
THIS IS A PROTEST FOR WHO IS SUPPORTING THE WAR	

DIFESA.IT - ENO7 - X-MASTER - CHERNOBILE - CODEXPLODER'tq

[illegible]

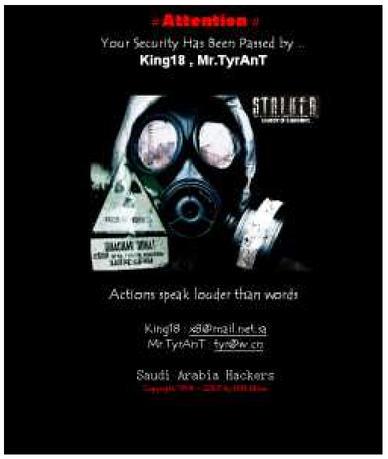
REVENGE OF THE TRABLUSGARP

TRABLUSGARP'IN INTIKAMI

[illegible]

[illegible] www.eno7.org

(a) Political/religious message



:: Attention ::

Your Security Has Been Passed by ..

King18 , Mr.TyrAnT

Actions speak louder than words

King18 : x8@mail.net.sa

Mr.TyrAnT : tyr@w.cn

Saudi Arabia Hackers

(b) Attacker signature

Figure 1.1: A sample of two different kinds of defacement content.

facements during year 2005, which corresponds to more than 1300 web pages successfully defaced each day [83]. Other salient numerical information are presented in Chapter 8. Web site defacement is included in the annual CSI/FBI Computer Crime and Security Survey, starting from its 9th edition (2004). According to the latest editions of this survey [22, 62], the defacement of web sites is one of the few attack categories whose trend is still growing and it "continues to plague organizations".

The practical relevance of this menace is also supported by the abundant and steadily expanding anecdotal evidence. Interesting defacements occurred recently include, e.g., those suffered by the Company Registration Office in Ireland, which was defaced in December 2006 and remained so until mid-January 2007 [69]; by the Italian Air Force, which was defaced on January 26th 2007 by a Turkish attacker protesting against "who is supporting the war" (this defacement is shown in Figure 1.1(a); by the United Nations [30] and by the Spanish Ministry of Housing [25] in August 2007. The list is much longer: [60, 34, 70, 48, 24, 15].

A defaced site typically contains only a few messages or images representing a sort of signature of the hacker that performed the defacement. A defaced site may also contain disturbing images or texts, political messages and so on: Figure 1.1 shows two real defacements which exhibit different kinds of content. Damages could also be not so explicit, however. An attacker capable of performing a complete replacement of a web resource would be also able to perform subtler changes difficult, if not impossible, for a user to point out—e.g., modifying a script that collects username and password sent by the user, so that these credentials are also sent to a location chosen by the attacker.

An attacker may commit a web defacement in many ways; yet, since we focused on detection rather than on prevention, the attack method issue is orthogonal to the main topic of this thesis. Nevertheless, the next section briefly discusses about typical ways in which a defacement is committed.

1.1.1 Attacks methods

From a general point of view, attacks to hosts and computer networks typically involves one or more of the following strategies.

- The attacker tries to convince the victim to perform some action that is indeed useful for the attacker himself; these actions may involve giving away some credentials that could be used later by the attacker in order to actually carry out the attack.
- The attacker is or collaborates with an insider intruder, i.e., a person within the organization being attacked, which can perform some key action or provide the former with sensible credentials.
- The attacker exploits one or more implementation errors, design errors or intrinsic limitations in the targeted system.

The last strategy is probably the most used by attackers. In particular, according to [83], the most popular way to commit a defacement involves exploiting a web application or web framework bug, for example by performing a script file inclusion, a SQL injection, a web server intrusion and so on.

1.2 Thesis outline

The remaining part of this thesis is organized as follows. Chapter 2 presents a review of the state of the art of web defacement detection and related topics. The available solutions for the specific problem and the most interesting recent advances in related fields are presented. In particular, the following fields are considered: storage integrity checkers, web similarity and change detection, intrusion detection and spam filtering. State of the art of the anomaly detection approach is also presented.

Chapter 3 describes the details about our proposal. It introduces the scenario, an overview of the framework and then we it goes down into the system architecture details; some interesting implementation detail are also covered. This and the following chapter discuss the main part of this work: design, analysis and implementation of an anomaly-based framework aimed at detecting web defacements, and, more broadly, unusual changes to web resources, without requiring any participation from the monitored site. This includes, in particular:

1. Techniques for building a profile of a dynamic web resource useful from the point of view of web defacement detection, i.e., the profile must be general enough to not generate an excessive number of false positives due to the dynamic nature of the resource, but, at the same time, specific enough to detect unusual changes.

2. Techniques for comparing a snapshot of a web resource to its profile. This problem involves a number of simple analyses and the aggregation of the corresponding results.

3. Policies for deciding when and how the profile of a web resource needs to be updated.

Chapter 4 presents the results of our deep experimental evaluation. It first describes the goals of the evaluation. Then, the methodology is presented, including the used dataset, the specific procedures leading the experiments and the indexes that have been measured. In particular, this chapter discusses about the effectiveness of the approach in terms of accuracy of detection—i.e., missed detections and false alarms—both in the short term and in the long term. The latter scenario importance is more evident due to the fact that web pages are dynamic: indeed results show that some kind of retuning of the tool is necessary in order to keep the profile updated. The timings of detection are also considered, both in terms of time needed in order to start the monitoring activity and in terms of promptness of detection—that is, how fast can be the detection of a defacement that has just been applied. Finally, this chapter focuses on scalability issues, with respect to a possible real deployment of the proposed framework: i.e., it has been figured out how many resources are needed in order to monitor a given number of remote web pages.

In Chapter 5 the problem of misclassified readings in the learning set is considered. The effectiveness of anomaly detection approach, and hence of the proposed framework, bases on the assumption that the profile is consistent with the monitored system. In particular, the assumption that the profile is computed starting from a "good" learning set—i.e., a learning set which is not corrupted by attacks—is often taken for granted. First, the influence of leaning set corruption on our framework effectiveness is assessed, in terms of possible wrong classifications. Then, a procedure aimed at discovering when a given unknown learning set is corrupted by positive readings is proposed and evaluated experimentally. This chapter also surveys the state of the art of corruption detection in anomaly detection and, more broadly, in classification and it discusses about the limitations a possible application of present approaches to the specific problem of web site defacement detection.

Chapter 6 describes our proposal to use Genetic Programming to perform the web site defacement detection task. Genetic Programming (GP) is an automatic method for creating computer programs by means of artificial evolution. The GP approach is used inside the proposed framework and experimentally evaluated on several web pages with varying dynamism, content and appearance.

Chapter 7 discusses the comparative evaluation of other approaches for defacement detection. A set of techniques that have been used in literature for designing several host-based or network-based Intrusion Detection Systems are considered. The results of a comparative experimental evaluation, using our approach as a baseline, are presented.

Finally, in Chapter 8 the findings of a large-scale study on reaction time to web site defacement are presented. There exist several statistics that indicate the number of incidents of this sort but there is a crucial piece of information still lacking: the typical *duration* of a defacement. Clearly, a defacement lasting one week is much more harmful than one of few minutes. A two months monitoring activity has been performed over more than 62000 defacements in order to figure out whether and when a reaction to the defacement is taken. It is shown that such time tends to be unacceptably long—in the order of several days—and with a long-tailed distribution.

Chapter 2

State of the art

This chapter describes the state of the art concerning defacement detection and related fields. First, existing tools specifically devoted to web defacement detection (Section 2.1) are presented, and then other tools that could be used for the same goal (Section 2.2). Next, works are surveyed that analyze similarities and differences in web content over the time (Section 2.3). Section 2.4 discusses the relation between our approach and intrusion detection, anomaly detection (Section 2.5) and, finally, spam detection (Section 2.6).

2.1 Tools for web defacement detection

Several tools for automatic detection of web defacements exist and some of them are commercially available. They are generally meant to be run *within* the site to be monitored (e.g., WebAgain http://www.lockstep.com/webagain). Some of these tools may be run remotely (e.g., Catbird http://www.catbird.com), but this is irrelevant to the following discussion.

All the tools that we are aware of are based on essentially the same idea: a copy (or baseline) of the resource to be monitored is kept in a "very safe" location; the resource content is compared to the *trusted copy* and an alert is generated whenever there is a mismatch. The comparison may be done periodically, based on a passive monitoring scheme, or whenever the resource content is about to be returned to a client. In the latter case, performance of the site may obviously be affected and an appliance devoted to carrying out the comparisons on-the-fly may be required. The trusted copy is usually stored in the form of a hash or digital signature of the resource originally posted, for efficiency reasons [64, 18]. Clearly, whenever a resource is modified, its trusted copy has to be updated accordingly.

The site administrator must be able to provide a valid baseline for the comparison and keep it constantly updated. Fulfilling this requirement may be difficult because most web resources are built on the fly dynamically, often by aggregating pieces of information from sources that may be dispersed throughout the organization and including portions that may be tailored to the client in a hardly predictable way—e.g., advertisement sections.

In order to simplify maintenance of the trusted copy, the site administrator may usually instruct the monitoring tool to analyze only selected (static) portions of each resource. This practice opens the door to unauthorized changes that could go undetected—the counterpart of unauthorized changes that, in our approach, might remain hidden within the profile.

Approaches based on a trusted copy of the resource to be monitored constitute perhaps the best solution currently available for automatic detection of web defacements, *provided* the organization may indeed afford to: (i) buy, install, configure, maintain the related technology; and (ii) integrate the technology with the daily web-related operational process of the organization—e.g., streaming the up-to-date resource content to the trusted copy. As already pointed out in the introduction, though, it seems fair to say that there are plenty of organizations which do not fulfill these requirements and consequently do not make use of these technologies. The result is that most organizations simply lack an automated and systematic surveillance of the integrity of their web resources. The approach proposed in this thesis attempts to address precisely such organizations. A surveillance service that can be joined with just a few mouse clicks and that have no impact whatsoever on the daily web-related operations would certainly have the potential to achieve a vast diffusion. It is important to remark once again that our proposal is meant to be an alternative to the trusted copy approach—for which there are currently *no* alternatives—and not a replacement for it.

2.2 Storage integrity checker

Web defacements may be detected also with tools not devoted specifically to such purpose. A *file system integrity checker*, for example, detects changes to portions of the file system that are not supposed to change, as well as deviations from a baseline content previously established by an administrator and kept in a safe place [33]. A more powerful and flexible approach is taken by *storage-based intrusion detection systems*, that also allow specifying an extensible set of update activities to be considered as suspicious [59, 4, 68, 21].

The analysis of the previous section may be applied to these tools as well: they must run within the site to be monitored, on carefully protected platforms and that could be difficult to deploy and maintain in settings where web resources aggregate pieces of information extracted from several sources, possibly other than the storage itself.

2.3 Web similarity and change detection

The tool that is proposed in this thesis analyzes a web resource based on its content and appearance. Similar analyses have been proposed for different purposes. SiteWatcher [44, 20] is a tool aimed at discovering *phishing* pages: it compares the original page against the potential phishing page and assesses visual similarities between them in terms of key regions, resource layouts, and overall styles. The analysis is mostly based on the visual appearance of the resource, whereas in the proposed framework is concerned also with attacks directed at resource features that could not affect the visible look of the resource. WebVigil [63] is a tool closer to this thesis proposal in this respect, since it considers many different resource features: it is a change-detection and notification system which can monitor a resource over the time in order to detect changes at different levels of granularity and then notify users who previously subscribed to such changes.

A fundamental difference with this proposal is that these tools do not base comparison on a profile of the resource. SiteWatcher simply compares the genuine web resource, which is supposed to be either reachable or locally available, to the suspected phishing resource. In WebVigil, the user himself defines in detail what kind of changes should be addressed by the comparison, which is then done between two consecutive versions of the resource. In the framework proposed in this work, a profile of the resource is first built automatically, then the resource, as currently available, is compared against its profile.

Important insights about the temporal evolution of web resources have been provided in [17]. The cited work presents a large-scale study, involving 151 millions of web resources observed once a week for 11 weeks. They used a similarity measure between resources based on the notion of shingle [7], a sequence of words excluding HTML markups (see the cited papers for details). Such a measure is quite different from what presented in this thesis, thus their results cannot be applied directly to our framework. However, that analysis does provide qualitative observations that are important from our point of view. In particular, they found that when changes to web pages occur, these usually affect only the HTML structure and do so in minor ways (addition/deletion of a tag and alike). This fact appears to imply that, broadly speaking, it is indeed possible to filter out automatically the dynamic changes of a resource while retaining its "essential" features. They also found that past changes to a resource are a good predictor of future changes. In other words, one may infer the degree of dynamism of a resource, at least in the short-term, by observing the resource for some period. From the point of view of this work, this property is important for "tightening" the profile appropriately.

The above results are confirmed by [55], which observed 150 highly popular web sites during one year. This work uses yet another similarity measure between resources (based on an order-independent analysis of the textual content, which is most meaningful from a search engine point of view), hence the results have to be interpreted with the same care as above. This study confirms that most resources change in a highly predictable way and that the past degree of change is strongly correlated with the future degree of change. It also highlights other features that are crucial to our framework. First, the correlation between past behavior and future behavior can vary widely from site to site and short-term prediction is very challenging for a non negligible fraction of resources. These observations confirm the (obvious) intuition that a one-size-fits-all approach cannot work in our case: change detection should be based on a profile that is tailored individually to each resource. Second, the ability to predict degree of change degrades over time. This means that a profile should probably be refreshed every now and then in order to remain sufficiently accurate.

2.4 Intrusion detection

An Intrusion Detection System (IDS) attempts to detect signs of unauthorized access to a monitored system by analyzing some part of the system itself. The analysis is typically based on a subset of the *inputs* of the monitored system. The nature of the inputs depends on the nature of the IDS: system call sequences in Host-based IDSs (e.g., [54, 11]), network traffic in Network-based IDSs (e.g., Snort http://www.snort.org, [10, 26]), application messages or events in Application Protocol-based IDSs (e.g., [37, 2]). The approach proposed in this thesis analyzes instead the *state* of the monitored system. Unlike storage-based IDS and storage integrity checkers (see Section 2.2) that access the internal state, moreover, it analyzes the externally observable state.

Working on the inputs has the potential to detect any malicious activity promptly and even prevent it completely. In practice, this potential is quite hard to exploit due to the large number of false alarms generated by tools of this kind. Working on the external state, on the other hand, may only detect intrusions after they have occurred. An approach of this kind, thus, makes sense only if it exhibits very good performance—low false negative rate, low false positive rate—and may support a monitoring frequency sufficiently high. The findings of this thesis in this respect, as resulting from the experimental evaluation, are very encouraging. Interestingly, an intrusion detection tool that observes the external state has the freedom of selecting the monitoring frequency depending on the desired trade-off between quickness of detection, available resources (computing and operators), priority of the monitored resources. A tool that observes the inputs must instead work at the speed imposed by the external environment: the tool must catch every single input event, since each one could be part of an attack.

There have been several proposals for analyzing a stream of input events by constructing a form of state machine driven by such events (e.g., [49, 65]). The notion of "state" in such approaches is quite different from what presented in this thesis. Here, the externally observable state of the system is considered, rather than on the state of an automaton constructed in a training phase. From this point of view this thesis approach is more similar to some emerging techniques for invariant-based bug detection. The technique described in [13] consists in inspecting periodically the content of the heap during program execution—i.e., part of the *internal* state. The values of certain metrics computed on such content are compared to a (program-specific) profile previously built in a training phase. Any anomaly is taken as an indication of a bug. The frequency of inspection of the state may be chosen at will, depending on how fast one would like to detect anomalies (rather than depending on how fast new inputs arrive).

2.5 Anomaly detection

This thesis work borrows many ideas from *anomaly-based* IDSs [16, 23, 36, 81]. Broadly speaking, a system is observed to learn its behavior and an alert is raised whenever something unusual is detected. Clearly, the technical details are very different: rather than observing network traffic or system call invocations, web resources are observed.

A framework able to detect anomalous system call invocations is proposed in [54]. For a given application and a given system call, the tool tries to characterize the typical values of the arguments of the call. This is done by constructing different *models* for the call, each focussed on a specific feature of its arguments (string length, char distribution, and so on) during a learning phase. During the monitoring phase, the tool intercepts each invocation and compares its arguments against the corresponding models. The results of the comparison are combined to produce a binary classification (normal or anomalous) of the evaluated system call. The behavior of a given application on a given healthy system, in terms of system call invocations, is supposed not to change over the time, thus there is no need to update the models after they have been constructed. This approach does not work in the scenario considered in this thesis, hence the problem of finding an algorithm for deciding when and how to update the profile of a resource has been faced.

A system able to detect web-based attacks was proposed earlier by the same authors, along very similar lines. For a given web application, the system proposed in [37] builds a profile of HTTP requests directed to that application. Then, after the learning phase has completed, the tool raises an alert whenever a request is suspected to be anomalous. The system may be used to detect attacks exploiting known and unknown vulnerabilities, including those which could enable applying a defacement. Like in the previous case, profile updating is not treated.

Two comparative studies of several anomaly detection techniques for intrusion detection are provided by [41] and [58], while techniques based on statistical classification are presented in [67, 77]. All these techniques map each observed event to a point in a multidimensional space, with one dimension for each analyzed feature. A profile is a region of that space and each point outside of that region is treated as an anomaly. In the main proposal presented in this thesis, events (reading of a web resource) have not been interpreted as a points in a multidimensional space, because it has not been found a satisfactory method for giving an uniform interpretation to very heterogeneous pieces of information—e.g., byte size, relative frequencies of HTML elements, fraction of missing recurrent images (see also Section 3.3.1).

The profile used as baseline for defining anomalies is usually defined in a preliminary learning phase and then left unchanged. An exception to this general approach may be found in [66], which describes a system for host intrusion detection in which the profile of the user is retuned periodically. In the scenario of web site defacement detection this form of retuning has turned out to be a necessity and thus constitutes an essential component of our approach.

2.6 Spam filtering

This thesis approach to web defacement detection exhibits interesting similarities to *spam detection*, i.e., the problem of separating unsolicited bulk electronic messages from legitimate ones. In both cases, one wants to classify an item as being "regular" or "anomalous" without defining in detail how the two categories should look like. In these terms, it is the act of blindly mass-mailing a message that makes it spam, not its actual content; similarly, it is the act of fraudulently replacing the original resource that makes the new one a defacement. Put it this way, any solution attempt that looks only at the content of a single mail or web resource could appear hopeless. Nevertheless, tools for spam filtering (labeling) are quite effective—e.g., SpamAssassin or Brightmail. Such tools exploit signatures for the content of spam messages and may have to update such signatures to follow new trends in spammers' behaviors [14]. Recently, more sophisticated approaches—e.g., machine learning—have been proposed to cope with the spam problem [1].

Broadly speaking, spam filtering differs from web defacement detection in the cost of "falses", i.e., wrong classifications. Most users perceive a rejected legitimate message (false positive) as a severe damage, while they do not care too much if some spam message is wrongly classified as legitimate (false negative). On the contrary, a missed web defacement may be much more costly than a false positive.

An interesting related problem that has emerged recently is *post spam* (also called *link spam*) detection. This problem affects web resources that are freely editable by users—e.g., blog and wiki resources. Attackers insert posts containing links to unrelated sites, often with automatic agents, in order to affect the effectiveness of search engines based on ranking algorithms that use link analysis—e.g., PageRank [56]. A framework for detecting spam posts in blog resources is proposed in [50]: the authors build a language model for the whole blog resource, one for each post and one for its linked resource, if any. Then, they compare these models and classify each post accordingly. This approach requires neither hard-coded rules nor prior knowledge of the monitored blog and evaluates items based on a global context, similarly to our approach. The main difference with this thesis approach is in the nature of the profile: they focus on a single, sophisticated, feature based on the textual content of the resource, whereas here a number of simple and easy to construct features are considered.

Chapter 3

Our approach

In this chapter we give some terminology and present our approach: first, from a general point of view in Section 3.1 and 3.2; then, we describe our prototype in detail in Section 3.3 and 3.4.

3.1 Key idea and design goals

We proposed a tool, that we call *Goldrake*, capable of checking the integrity of many *remote* web resources automatically. We wanted to achieve a fundamental design goal: the monitoring tool should not have required any participation from the monitored site. This feature constitutes also the novelty of our approach in respect to similar existing technology for detecting web site defacements. In particular, Goldrake does not require the installation of any infrastructure at the monitored site, nor does it require the knowledge of the "officially approved" content of the resource and nor does it require any change in the operational procedures of web managing by the organization which own the monitored site.

The above-mentioned features, joined with the fact that the tool is designed for monitoring remote web pages, candidates the proposed approach for building a remote, large-scale, automatic *monitoring service*. Such a service, which structure is outlined in Figure 3.1, may be joined with just a few mouse clicks and does not impose any burden on the normal operation of a web site, and hence would have the potential to be really used widely. Thereby, it could greatly improve robustness and dependability of the web infrastructure. We remark however that our proposal is not meant to *replace* the existing approach, which is based on a *trusted copy* of the page to be monitored. We merely aim to propose a different option, which incorporates a different trade-off in terms of operational and maintenance costs, ease of use and accuracy of detection. We also note that there is currently no alternative to the trusted copy approach.

Our approach is based on *anomaly detection*. During a preliminary learning phase Goldrake builds automatically a profile of the monitored resource. Then, while monitoring, Goldrake will retrieve the remote resource periodically and generate an alert

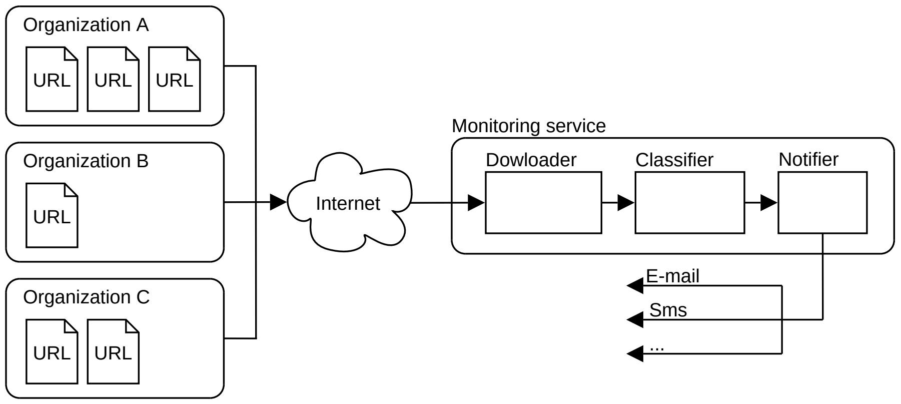


Figure 3.1: The monitoring service: one single instance is able to monitor several remote organizations web pages.

whenever something "unusual" shows up.

Implementing this simple idea is hard because web content is highly dynamic, that is, different readings of the same web resource may be very dissimilar from each other. Moreover, extent and frequency of individual changes may vary widely across resources [55]. The challenge, thus, is how to develop a tool capable of dealing with highly dynamic content while keeping false positives to a minimum and, at the same time, while generating meaningful alerts. That is, alerts notifying changes so unusual to deserve further analysis by a human operator.

Clearly, the tool must be tuned for each specific resource, because a one-size-fits-all combination of parameters cannot work. The problem is complicated further by our desire to address a *large-scale* scenario in which the service may start monitoring each newly added resource *quickly*—after a learning phase of just a few days at most. Large-scale implies that the tool should tune itself automatically without requiring the involvement of human operators, otherwise scalability would be severely affected. A short learning phase implies that the profile of the resource may hardly be fully complete or fully accurate.

A key rationale for our framework is that it analyzes many features from many points of view, basing on the following considerations. We realized from the early experiments that we could achieve a remarkable accuracy by combining a large number of simple analyses, rather than by focusing on a few more complex criteria. We also realized that a given feature should be analyzed from several points of view—e.g., a given link may be absolute or relative; the pointed URL may contain unusual words or not; it may appear in all readings or only in some of them and so on.

3.2 Scenario

We call a *web resource*, resource for short, a piece of data that is univocally identified by an URL—e.g., an HTML document, an image file, and so on. In practice, in this study, we focused mostly on web pages—i.e., HTML documents including images, style sheets, JavaScripts and so on—but we considered also different resources—e.g., RSS (Really Simple Syndication) feeds.

A *monitoring service* M can monitor several different web resources at the same time. The *monitored set* is the set of URLs identifying the monitored resources. Typically, but not necessarily, the monitored set will include many resources from many different (remote) organizations. For ease of presentation but without loss of generality, in the following reasonings, we often assume that the monitored set contains only one resource R. We denote by r_i the snapshot or *reading* of R at time t_i. We will omit the subscript when not required.

In a first phase, which we call the *learning phase*, M builds the *profile* of R, denoted by P_R. To this end, M collects a sequence $\{r_1, r_2, \dots\}$ of readings of R, that we call the *tuning sequence*, which is actually the learning set. Then, M builds P_R by applying a *tuning procedure* on the tuning sequence. Having completed the learning phase, M may enter the *monitoring phase* in which it executes the following cycle in an endless way:

1. wait for a specified *monitoring interval* m;
2. fetch a reading r of R;
3. classify r by analyzing it against its profile P_R;
4. if r appears to be unusual, then execute some action.

The discussion of our prototype includes only learning phase and analysis (step 3 above). The other steps of the monitoring phase are performed by other non-key components of the service (see Figure 3.1) and can be understood easily. We point out, in particular, that the actual implementation of steps 2—i.e., how to retrieve the current reader—and 4—i.e., how to send an alert to the monitored site and, at that site, how to handle the alert—are orthogonal to the topic of this thesis. In a real deployment scenario, the tool could notify the administrator of a defaced site with an email, a SMS, and so on.

3.3 System architecture

The core component of the monitoring service has to classify each reading as being either *normal* or *anomalous*. In practice, this component is a binary classifier which consists internally of a *refiner* followed by an *aggregator*: Figure 3.2 shows the internal architecture of the classifier.

The refiner implements a function that takes a reading r and produces a fixed size numeric vector $v = \mathcal{R}(r) \in \mathbb{R}^n$. In our case the transformation involves evaluating and quantifying many features of a web page related to both its content and appearance (all details are given in Section 3.3.1). The refiner is internally composed by one or more *sensors*. A sensor S is a component which receives as input the reading r and outputs a fixed size vector of real numbers v_S. The output of the refiner is composed

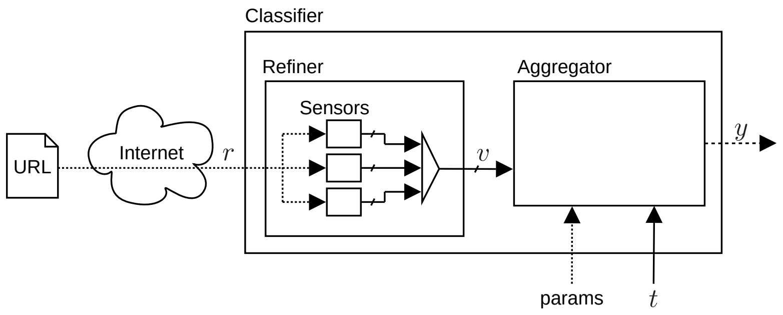


Figure 3.2: Our binary classifier architecture. Different arrow types correspond to different types of data.

by concatenating the output of all sensors. Sensors are functional blocks and have no internal state: $v = \mathcal{R}(r)$ depends only on the current input r and does not depend on any prior reading. The refiner produces a vector $v = \mathcal{R}(r)$ of 1466 elements, obtained by concatenating the outputs from 43 different sensors (Section 3.3.1): we denote the output of the refiner for reading r_k by v_k.

The aggregator is the core component of the detector and it is the one that actually implements the anomaly detection. In a first phase, that we call the *learning phase*, the aggregator collects a number of readings in order to build the profile of the resource; during this phase, the aggregator is not able to classify readings. In a second phase, the *monitoring phase*, the aggregator compares the current reading against the profile and considers it as anomalous whenever it is too much different from the profile. The output in the monitoring phase depends also on an external parameter called *normalized discrimination threshold* (threshold for short), which affects the sensitivity-specificity tradeoff of the detector. We denote the threshold by t.

The aggregator performs a *tuning procedure* at the end of the learning phase on a tuning sequence obtained with the first l readings. With this procedure the aggregator sets the values for some internal numeric parameters, that constitute the profile P_R of the resource R. During the learning phase, the output y_k for each reading r_k is always $y_k = \text{UNABLE}$, meaning that the aggregator is currently unable to classify the reading.

After the learning phase, the aggregator enters the *monitoring phase*, in which it considers the remaining readings. In this phase the aggregator compares each reading against the profile P_R established in the learning phase. The output y_k for each reading r_k is given by a function $F^A_{\text{compare}}(v_k, P_R, t)$ that may return either $y_k = \text{NEGATIVE}$ (meaning the reading is normal) or $y_k = \text{POSITIVE}$ (meaning the reading is anomalous).

We built many different aggregators, which differ in the way they exploit application-specific knowledge and in the techniques they internally adopt in order to elaborate the outputs produced by the refiner. The details are presented in Section 3.3.2 and in Chapters 6 and 7.

As an aside, the early internal architecture of the system was quite different from

Table 3.1: Sensor categories and corresponding vector portion sizes.

Category	Number of sensors	Vector size
Cardinality	25	25
RelativeFrequencies	2	117
HashedItemCounter	10	920
HashedTree	2	200
Signature	4	4
Total	43	1466

what described above. Sensors were dynamic systems which did hold the profile inside them; in other words, the output of a sensor for a given reading did depend also on previous readings. Each sensor contained the portion of the profile related to the analysis it performed; then, during the monitoring activity, the sensor itself returned a boolean value representing its verdict about the current reading. The aggregator, on the contrary, was quite simple: it combined the sensor outputs with a simple boolean function and produced the final output. Note, however, that the key idea basing the proposed approach remains the same. The reasons because we designed a different framework concerns efficiency—mainly in terms of performance, but also in terms of easiness of deployment and code maintenance—and the possibility to evaluate different approaches (like we indeed did, see Chapters 6 and 7). In this thesis, we only discuss the latter version of the framework.

3.3.1 Sensors

The 43 sensors contained in our refiner can be divided in 5 categories, based on the way they extract information from readings. A brief description of each category (or *group*) follows. Table 3.1 summarizes salient information about sensor categories and indicates the number of sensors and the corresponding size for the vector v portion in each category.

Cardinality sensors

Each sensor in this category outputs a vector composed by only 1 element, i.e., $v_S = v^1$. The value of v^1 corresponds to the measure of some simple feature of the reading (e.g., the number of lines).

The features taken into account by the sensors of this category are:

- Tags: block type (e.g., the output v^1 of the sensor is a count of the number of block type tags in the reading), content type, text decoration type, title type, form type, structural type, table type, distinct types, all tags, with `class` attribute;
- Size attributes: byte size, mean size of text blocks, number of lines, text length;

- Text style attributes: number of text case shifts, number of letter-to-digit and digit-to-letter shifts, uppercase-to-lowercase ratio;
- Other items: images (all, those whose names contain a digit), forms, tables, links (all, containing a digit, external, absolute).

RelativeFrequencies sensors

Each sensor S in this category outputs a vector composed by n_S elements, i.e., $v_S = \{v^1, \dots, v^{n_S}\}$. Given a reading r, S computes the relative frequency of each item in the item class analyzed by S (e.g., lowercase letters), whose size is known and equal to n_S. The value of the element v^k is equal to the relative frequency of the k-th item of the given class.

This category includes two sensors. One analyzes lowercase letters contained in the visible textual part of the resource ($n_S = 26$); the other analyzes HTML elements of the resource—e.g., `html`, `body`, `head`, and so on—with $n_S = 91$.

For example, consider the sample HTML document of Listing 3.1 and the Relative-Frequencies sensor working on HTML elements. There are 15 different HTML elements (in order of appearance: `html`, `head`, `title`, `link`, `body`, `h1`, `p`, `a`, `b`, `table`, `tr`, `th`, `td`, `br` and `img`) accounting for 25 start tags (end tags are not counted). The `html` element appears only one time, hence the corresponding element of the vector v_S is equal to $1/25 = 0.04$; `p` appears 4 times, hence its relative frequency is 0.16; and so on. The vector v_S will be composed by 15 non-zero values and $91 - 15 = 76$ elements equal to 0.

HashedItemsCounter sensors

Each sensor S in this category outputs a vector composed by n_S elements, i.e., $v_S = \{v^1, \dots, v^{n_S}\}$ and works as follows. Given a reading r, the sensor S:

1. sets to 0 each element v^k of v_S;
2. builds a set $L = \{l_1, l_2, \dots\}$ of items belonging to the considered class (e.g., absolute linked URLs) and found in r; note that L contains no duplicate items;
3. for each item l_j, applies a hash function to l_j obtaining a value $1 \leq k_j \leq n_S$;
4. increments v^{k_j} by 1.

This category includes 10 sensors, each associated with one of the following item classes: image URLs (all images, only those whose name contains on or more digits), embedded scripts, tags, words contained in the visible textual part of the resource and linked URLs. The link feature is considered as 5 different sub-features, i.e., by 5 different sensors of this group: all external, all absolute, all without digits, external without digits, absolute without digits. All of the above sensors use a hash function such that $n_S = 100$, except from the sensor considering embedded scripts for which $n_S = 20$. Note that different items could be hashed on the same vector element. We use a large vector size to minimize this possibility, which cannot be avoided completely, however.

Listing 3.1: A simple HTML document.

```html
<html>
  <head>
    <title>Simple sample page</title>
    <link type="text/css" rel="stylesheet" title="Default" href="css/
       default.css">
  </head>
  <body>
    <h1>Simple samples</h1>
    <p>There is some <a href="someplace.html">linked</a> sample text.
       </p>
    <p>There is some <b>decorated</b> text.</p>
    <h1>Complex samples</h1>
    <p>A table:</p>
    <table>
      <tr><th>Name</th><th>Number</th></tr>
      <tr><td>John Smith</td><td>123</td></tr>
      <tr><td>Jack Ripper</td><td>456</td></tr>
    </table>
    <p>And a figure:<br>
      <img src="images/sample.png">
    </p>
  </body>
</html>
```

For example, consider the sample HTML document of Listing 3.1 and the HashedItemsCounter sensor working on words contained in the visible textual part of the page. The word `some` appears 2 times, while the word `Jack` appears only once. Hence, the element of vector v_S whose index is equal to the output of the hash function applied to `some` (e.g., v_S^{38}) will incremented by 2; the element of vector v_S corresponding to `Jack` (e.g., v_S^{76}) will incremented by 1. Suppose that no other word in the sample document is such that its hash function is either 38 or 76, then $v_S^{38} = 2$ and $v_S^{76} = 1$. If, otherwise, the hash of the word `there` is, for example, equal to 38, then $v_S^{38} = 4$, since `there` appears twice.

HashedTree sensors

Each sensor S in this category outputs a vector composed by n_S elements, i.e., $v_S = \{v^1, \dots, v^{n_S}\}$ and works as follows. Given a reading r, S:

1. sets to 0 each element v^k of v;
2. builds a tree H by applying a sensor-specific transformation on the HTML/XML tree of r (see below);
3. for each node $h_{l,j}$ of the level l of H, applies a hash function to $h_{l,j}$ obtaining a value $k_{l,j}$
4. increments $v^{k_{l,j}}$ by 1.

The hash function is such that different levels of the tree are mapped to different adjacent partitions of the output vector v, i.e., each partition is "reserved" for storing information about a single tree level.

This category includes two sensors, one for each of the following transformations:

- Each start tag node of the HTML/XML tree of reading r corresponds to a node in the transformed tree H. Nodes of H contain only the type of the tag, i.e., the HTML element (for example, `table` could be a node of H, whereas `<table class="name">` could not).
- Only nodes of the HTML/XML tree of reading r that are tags of a predefined set (`html`, `body`, `head`, `div`, `table`, `tr`, `td`, `form`, `frame`, `input`, `textarea`, `style`, `script`) correspond to a node in the transformed tree H. Nodes of H contain the full start tag (for example, `<td class="name">` could be a node of H, whereas `<p id="news">` could not).

Both sensor have $n_S = 200$ and use $2, 4, 50, 90$ and 54 vector elements for storing information about respectively tree levels $1, 2, 3, 4$ and 5; thereby, nodes of level 6 and higher are not considered.

For example, consider the sample HTML document of Listing 3.1 and the HashedTree sensor working on the start tag nodes of the HTML/XML tree. The transformation produces the tree which is shown in Figure 3.3: note that this tree has a depth of 5

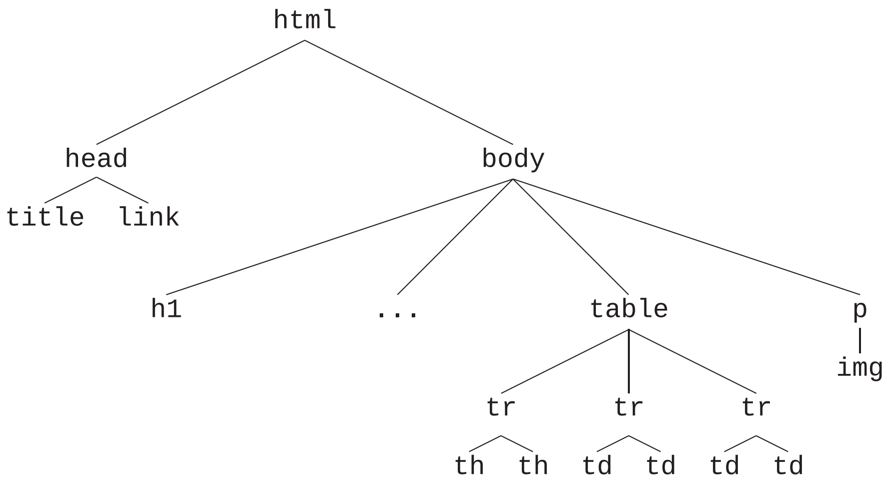


Figure 3.3: The result of a tree transformation of the document of Listing 3.1.

levels, hence no information are discarded. Since 2 elements of v_S—i.e., v_s^1 and v_S^2—are reserved for level 1 and level 1 contains only one node, one between v_s^1 and v_S^2 will be equal to 1, the other will be 0. Concerning the third level, there are 5 different HTML elements (`title`, `link`, `h1`, `p` and `table`): `p` appears 4 times, `h1` 2 times and the others one time. Hence, assuming that the hashes of the 5 considered elements are not equals, among the 50 elements of v_S corresponding to the third level (i.e., $v_S^7, \dots, v_S^{56}$), 3 of them will be equal to 1, one will be 2, one 4 and the remaining will be 0.

Signature sensors

Each sensor of this category outputs a vector composed by only 1 element v^1, whose value depends on the presence of a given attribute. For a given reading r, $v^1 = 1$ when the attribute is found and $v^1 = 0$ otherwise.

This category includes 4 sensors, one for each of the following attributes (rather common in defaced web pages):

- has a black background;
- contains only one image or no images at all;
- does not contain any tags;
- does not contain any visible text.

3.3.2 Aggregators

For the sake of the first prototype of Goldrake, we built 3 different aggregators. They exploit different levels of application-specific knowledge and are described in the next sections. Recall that the tuning sequence consists of a sequence $S_{\text{tuning}} = \{v_1, \dots, v_l\}$

obtained from the first l readings, where each v_k is a vector with 1466 elements. Other aggregators are presented in Chapters 6 and 7.

TooManyFiringElements

This aggregator does not exploit any application-specific knowledge. In brief, a reading is labeled as anomalous whenever too many elements of v_k are too much different from what expected.

In the tuning procedure the aggregator computes the mean η_i and standard deviation σ_i for each element v_k^i, across all vectors in $S_{\text{tuning}} = \{v_1, \ldots, v_l\}$. During the monitoring phase the aggregator counts the number of *firing elements*, i.e., those elements whose value is too much different from what expected. An element fires when its value v^i is such that $|v^i - \eta_i| \geq 3\sigma_i$. If the number of firing elements is at least Nt, the reading is classified as anomalous ($N = 1466$ is the size of each vector, t is the threshold).

Note that this aggregator handles all vector elements in the same way, irrespective of how they have been generated by the refiner. Thus, for example, elements generated by a signature sensor are handled in the same way as those generated by hashed. Moreover, the aggregator does not consider any information possibly associated with pairs or sets of elements, i.e., elements generated by either the same sensor or by sensors in the same category.

TooManyFiringSensors

This aggregator exploits some degree of domain-specific knowledge: it "knows" that the vector elements are partitioned in slices and each slice corresponds to a specific sensor. The profile constructed in the learning phase is also partitioned, with one partition associated with each sensor.

In the monitoring phase this aggregator transforms each slice in a boolean, by applying a sensor-specific transformation that depends on the profile (i.e., on the partition of the profile associated with that sensor). When the boolean obtained from a slice is true, we say that the corresponding sensor *fires*. If the number of sensors that fire is at least Mt, the reading is classified as anomalous ($M = 43$ is the number of sensors, t is the threshold).

We describe the details of the tuning procedure and monitoring phase below. All sensors in the same category are handled in the same way. As an aside, note that not only this aggregator exploits domain-specific knowledge, it also exploits knowledge about the refiner (e.g., regarding the number of sensors and size of each slice).

Cardinality In the tuning procedure the aggregator determines mean η and standard deviation σ of the values $v_1^1, \ldots, v_l^1$—recall that Cardinality sensors output a vector composed by a single value. In the monitoring phase a sensor fires if its output value v^1 is such that $|v^1 - \eta| \geq 3\sigma$.

$$
\begin{array}{rlllllllll}
v_{S,1} = \{ & 0.23 & 0.11 & 0.05 & 0.37 & 0.00 & 0.24 & \} & d_1 = & 0.06 \\
v_{S,2} = \{ & 0.21 & 0.13 & 0.08 & 0.39 & 0.00 & 0.19 & \} & d_2 = & 0.08 \\
v_{S,3} = \{ & 0.19 & 0.09 & 0.07 & 0.38 & 0.00 & 0.27 & \} & d_3 = & 0.11 \\
v_{S,4} = \{ & 0.23 & 0.11 & 0.04 & 0.42 & 0.00 & 0.20 & \} & d_4 = & 0.09 \\
\eta = \{ & 0.215 & 0.11 & 0.06 & 0.39 & 0.00 & 0.225 & \} & & \\
\multicolumn{10}{c}{\eta_d = 0.085,\ \sigma_d = 0.021} \\
v_S = \{ & 0.02 & 0.14 & 0.08 & 0.12 & 0.62 & 0.02 & \} & d = & 1.34 \\
\multicolumn{10}{c}{|d - \eta_d| = |1.34 - 0.085| = 1.255 > 0.063 = 3 \cdot 0.021}
\end{array}
$$

Figure 3.4: An example of a RelativeFrequencies sensor. See the text for the explanation.

RelativeFrequencies A sensor in this category fires when the relative frequencies (of the class items associated with the sensor) observed in the current reading are too much different from what expected. In detail, let n_S be the size of the slice output by a sensor S. In the tuning phase, the aggregator performs the following steps:

1. evaluates the mean values $\{\eta^1, \dots, \eta^{n_S}\}$ of the vector elements associated with S;
2. computes the following for each reading v_k of the tuning sequence ($k \in [1, l]$):
$$d_k = \sum_{i=1}^{n_S} |v_k^i - \eta^i| \tag{3.1}$$
3. computes mean η_d and standard deviation σ_d of $\{d_1, \dots, d_l\}$.

In the monitoring phase, for a given reading v, the aggregator computes:

$$d = \sum_{i=1}^{n_S} |v^i - \eta^i| \tag{3.2}$$

The corresponding sensor fires if and only if $|d - \eta_d| \geq 3\sigma_d$.

For example, consider the example shown in Figure 3.4: in this case, the size of the portion of v corresponding to this RelativeFrequencies sensor is $n_S = 6$ and the length of the tuning sequence is $l = 4$. Recall that each $v_{S,i}$ is a vector representing the relative frequencies of a given item class in the i-th reading, hence is such that $\sum_k v_{S,i}^k = 1$. The profile related to this sensor is composed by the vector η and the values η_d and σ_d. The example sensor fires, because, for the considered reading whose corresponding v_S is shown in the figure, $d - \eta_d$ is grater than $3\sigma_d$.

HashedItemsCounter Let n_S be the size of the slice output by a sensor S. In the tuning procedure, the aggregator computes for each slice element the minimum value across all readings in the tuning sequence, i.e., $\{m^1, \dots, m^{n_S}\}$. In the monitoring phase S fires if and only if at least one element v^i in the current reading is such that $v^i < m^i$.

The interpretation of this category is as follows. Consider a sensor S of this category and suppose that $n_S \to \infty$ (see also the description of sensors of this category in

Section 3.3.1). In this case, the size of the slice of the vector v output by S tends to ∞ too; besides, the hash function used by S maps each different element l of the class considered by the sensor to a precise element v^i of v: $v^i = 0$ if the given element is not present in the considered reading, $v^i = 1$ otherwise. It follows that the corresponding minimum value m^i is 1 if and only if the given element l was present in every reading included in the tuning sequence, i.e., if l was a "recurrent items". During the monitoring phase, S fires if there is at least one l for which the corresponding v^i is lower than m_i, i.e., if there is at least a missing recurrent item. Provided that n_S is obviously finite, the higher n_S, the lower the number of different items mapped to the same v^i and hence the more similar the real behavior to the description given above.

HashedTree Sensors in this category are handled in the same way as those of the previous category, but the interpretation of a firing is slightly different. Any non-zero element in $\{m^1, \dots, m^{n_S}\}$ corresponds to a node which appear in every reading of the tuning sequence, at the same level of the tree. In the monitoring phase the sensor fires when a portion of this "recurrent tree" is missing from the current reading (i.e., the sensor fires when the tree corresponding to the current reading is not a supertree of the recurrent tree). We omit further details for simplicity, as they can be figured out easily.

Signature A sensor in this category fires when its output is 1. Recall that these sensors output a single element vector, whose value is 1 whenever they find a specific attribute in the current reading.

TooManyFiringCategories

This aggregator works similarly to the previous one. It transforms slices into boolean values in the same way as above. However, rather then considering all sensors as being equivalent, this aggregator "knows" that sensors are grouped in categories. If the number of categories with at least one sensor that fires is at least Kt, the reading is classified as anomalous ($K = 5$ is the number of categories, t is the threshold).

The reason because we decided to group sensors outputs in categories for this aggregator, is that we noticed that sensors belonging to the same category tend to exhibit a similar behavior in terms of false positives. A legitimate, but substantial, modification in a resource may cause several sensors in the same group to fire simply because all such sensors perform their analysis in a similar way. For example, consider a resource containing a "news" section composed of a set of news items, each containing text, titles, images and so on. If the administrator of that site decides to increment the number of news items, it is likely that several Cardinality sensors will fire. Yet, since the overall structure of the resource does not change, neither HashedTree sensors nor HeshedItemCounter sensors will fire. This fact is supported by the experimental evaluation we performed, which is discussed in Section 4.8. This aggregator thus exploits domain-specific knowledge more deeply than the previous one.

3.3.3 Retuning policies

In our earlier experiments (see Section 4.4), we found that detection effectiveness of the aggregators described above remarkably decrease as time goes by. This occurs due to the fact that the profile of the monitored resource is no longer representative of its current content and appearance. In order to cope with this problem, we enabled our framework to perform some form of profile retuning, accordingly to a predefined *retuning policy*.

In particular, we defined 4 different retuning policies. Each policy specifies the conditions that trigger the execution of a tuning procedure. Note that all policies described in this section, except the last one, can be applied with all the aggregators described in Section 3.3.2. Due to the nature of the event that triggers the retuning, the last policy can only be applied in combination with the TooManyFiringCategories aggregator.

- *FixedInterval.* The tuning procedure is applied every n readings. This policy can be applied automatically, without any human intervention.
- *UponFalsePositives.* The tuning procedure is applied whenever the aggregator raises a false alarm. In other words, whenever the aggregator outputs a positive value there must be an human operator that analyzes the corresponding reading and decides whether this is indeed a true alarm or not. If not, Goldrake will make the aggregator execute the tuning procedure before analyzing the next reading.
- *Mixed.* This policy is a combination of the two previous ones, that is, the tuning procedure is applied when at least one of the following occurs:
  a. the aggregator raises a false alarm (labeled as such by a human operator);
  b. there have been n consecutive readings without any false alarm.
- *SelfVerifying.* The tuning procedure is applied when at least one of the following occurs:
  a. there are exactly $Kt - 1$ firing groups (recall that the TooManyFiringCategories aggregator fires if at least Kt groups fire)
  b. there have been n consecutive readings since the last retuning triggered by this condition.

  This policy can be applied automatically, without any human intervention. For example, when $t = 0.8$, the aggregator will perform a retuning if exactly 3 groups fire.

3.4 Implementation details

Our prototype, Goldrake, is written in Java and uses a relational database to store readings and other data. We paid special attention to make our tool *modular* and *extensible*. Sensors and aggregators are simply classes that implement a given interface; thereby,

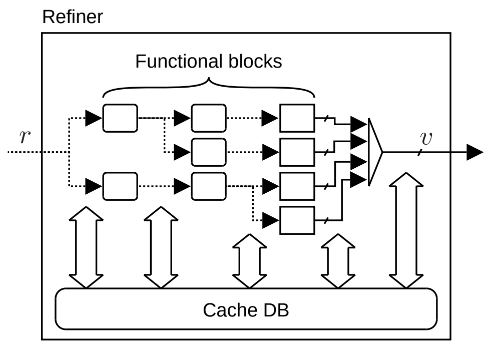


Figure 3.5: A representation of the internal structure of the refiner implementation (schematic). Different arrow types correspond to different types of data. Sensors elaboration is actually performed in several steps by functional blocks. Each block outcome is stored in the cache and hence can be retrieved from there without being computed more than once.

one can add new sensors or aggregators by simply developing Java classes implementing the appropriate interface. This is a key feature both for experimenting with the tool and for its production use and is indeed the way we followed to experiment with other approach variants (see Chapter 6 and 7).

We also took care to design internal architecture of Goldrake in order to obtain satisfactory performance, in terms of computation effort required to analyze a single reading; being a prototype, however, some space exists for further improvements. In this sense, we focused mainly on designing an efficient refiner: this issue is discussed in the next section.

3.4.1 Refiner

Almost all of the sensors perform several transformation steps in order to obtain a numeric vector starting from the input, i.e., the reading. Moreover, some of these steps are performed from many sensors. For example, many sensors execute the HTML parsing in order to obtain information about nodes, tags, and so on. We wanted to exploit this similarities among sensors in order to avoid computation redundancy, hence obtaining better performance in respect to the case in which some transformation step is performed twice.

We hence designed the implementation of the refiner as a layered graph of functional blocks. A simple graphical representation of a sample of such architecture is shown in Figure 3.5. Rightmost blocks use as input the reading itself; leftmost blocks output numeric vectors—these blocks indeed correspond to the sensors as described in Section 3.3.1. Each non rightmost block receives as input the output of one or more other blocks and, if needed, also the reading itself: if two or more blocks need the same piece of information—i.e., the output of the same block—it is computed only once. Thereby, during the evaluation of a reading by the refiner, no transformation steps are computed more than once. For example, consider the sensor belonging to the HashedItemCounter category which concerns words of visible textual part of a web page. Three main steps are performed:

1. the reading r is parsed, thus obtaining a data structure corresponding to the HTM-L/XML tree of the page;
2. the nodes of the tree that contains visible texts are selected and their content is extracted and concatenated
3. the text is split in words and, finally, the numeric vector v is generated as described in Section 3.3.1.

In practice, this optimization is obtained by caching in a relational database the output of each block for each readings. When the output of a given block is required by another block, it is not computed again, instead it is simply read from the cache.

Chapter 4

Experimental evaluation

4.1 Evaluation goals

Detecting when a web page has been defaced is substantially a classification task; in particular, it is a binary classification task. Hence, the effectiveness of the approach can be measured, in brief, as the number of correct readings correctly classified. To this end, we used the *False Positive Rate* (FPR) and the *False Negative Rate* (FNR) as effectiveness indexes: we describe them in a more detailed way in the next section. In both cases, the lower the index value, the more effective is the approach.

Due to the particular nature of the monitored systems (i.e., web pages), classification effectiveness can be influenced by several factors. In particular, we studied how the approach effectiveness is affected by web pages intrinsic dynamism (see Section 4.5) and by the presence of unexpected misclassified readings in the tuning sequence (see Chapter 5).

Beside classification effectiveness, we wanted also to evaluate the feasibility of the proposed approach from a broader point of view: number of pages that can be monitored with given hardware resources, minimum monitoring interval, and so on. In other words, we evaluated the *scalability* of the proposed approach: we define it more deeply in Section 4.9.

4.1.1 Effectiveness indexes

We evaluated the effectiveness of the proposed approach measuring false positives and false negatives. We say that the detector gives a *false positive* when it evaluates a normal (negative) reading as anomalous (positive)—i.e., a false alarm. We say that the detector gives a *false negative* when it evaluates an anomalous reading as normal—i.e., a missed detection of a defacement. In this chapter, we show experiments result in term of FPR an FNR: the former is the ratio between the number of false positives and the number of evaluated normal readings; the latter is the ratio between the number of false negatives and the number of evaluated anomalous readings.

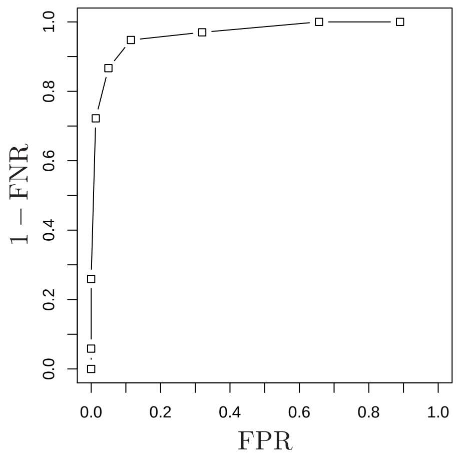


Figure 4.1: A ROC curve example.

While the goal is to obtain low FPR *and* FNR, usually one has to choose a tradeoff between low FPR and low FNR, perhaps adjusting some parameter—usually a *discrimination threshold*—of the detector being used. The desired tradeoff depends on the considered application and the specific deployment of that application. In our case, i.e., in the case of web site defacement detection, we think that it is more important to minimize FNR, that is, to minimize missed detection: a missed detection may be more costly than a false positive. This is different, for example, from spam detection where most users perceive a rejected legitimate message (false positive) as a severe damage, while they do not care too much if some spam message is wrongly classified as legitimate (false negative).

In the discussion that follows, we present some FPR results also in terms of False Positive Frequency (FPF), i.e., the number of false alarms per unit of time. This index is interesting because it should be more usable by human operators: an indication like "3 false alarms every week per resource" is simpler to understand and reason about than one like "3% of false alarms per resource". Moreover, such index is useful while elaborating on the approach expected scalability (see Section 4.9). Note that the possibility of using in practice the FPF index is a straightforward consequence of analyzing the state of the monitored system rather than its inputs.

Another effectiveness index that is quite popular among binary classifiers evaluation is the area under the ROC curve. The *Receiver Operating Characteristic*, or simply ROC curve, is the plot of the True Positive Rate (TPR $= 1 - \text{FPR}$) vs. the FPR as the classifier discrimination threshold is varied. Figure 4.1 show an example of a ROC curve. The nearer the curve stays to the upper left corner (where FPR $=$ FNR $= 0$), the better the classifier. The line joining the left bottom and the right top corners is the line of no discrimination, which corresponds to a random (weighted) guess used as a binary classifier. The ROC curve is plotted using many pairs $\langle \text{FPR}, \text{FNR} \rangle$ obtained testing the binary classifier on the same dataset with different value for the discrimination threshold.

The area under the ROC curve is an interesting effectiveness index which gives a synthetic evaluation of the binary classifier; differently from FPR and FNR indexes, it allows one to set aside from the possible actual value for the discrimination threshold, and hence from the selected tradeoff between false alarms and missed detections.

4.2 Dataset

We performed all our tests off-line, for convenience reasons, using an archive of readings that we obtained as follows; for some of the tests, we used only a portion of the archive. We observed 15 web pages for about 6 months, collecting a reading for each page every 6 hours, thus totaling about 650 readings for almost each web page. These readings compose the *negatives sequences*—one negative sequence S_N for each page: we visually inspected them in order to confirm the assumption that they are all genuine, that is, none of them was a defacement or an otherwise anomalous page (e.g., a server error page).

Table 4.1 presents a list of the observed pages, which includes pages of e-commerce web sites, newspapers web sites, and alike. Pages differ in size, content and dynamism. Almost all resources contain dynamic portions that change whenever the resource is accessed. In most cases such portions are generated in a way hardly predictable (including advertisements) and in some cases they account for a significant fraction of the overall content. For example, the main section of the Amazon – Home contains a list of product categories that seems to be generated by choosing at random from a predefined set of lists. The Wikipedia – Random page, shows an article of the free encyclopedia that is chosen randomly at every access. Most of the monitored resources contain a fixed structural part and a slowly changing content part, often including an ordered list of "items" (e.g., news, topics and alike). The frequency of changes may range from a few hours to several days. The impact of each change on the visual appearance of the resource may vary widely.

Unlike most works that evaluate IDSs [41, 43, 47], there is not a standard attack trace to use. We hence built a *positive sequence* S_P composed by 100 readings extracted from a publicly available defacement archive. Defacements composing S_P were not related with any of the 15 resources that we observed—as pointed out above none of these resources was defaced during our monitoring period. We chose attack samples with different size, language, layout and with or without images, scripts and other rich features.

4.3 Preliminary experiments

We performed some preliminary experiments which leaded us to the following findings. First, the TooManyFiringCategories aggregator is largely the best performing solution, among the three presented in Section 3.3.2. For this reason, in this chapter we only reason on this aggregator; nevertheless, a numerical confirmation of this finding can be found in Section 5.3.4. Second, we found that some kind of updating of the profile is necessary. We elaborate on this issue in the next sections.

Table 4.1: List of web resources composing our dataset. Change frequency is a rough approximation of how often non minor changes were applied to the resource, according to our observations. Concerning Amazon – Home page and Wikipedia – Random page, we noted that most of the content section of the resource changed at every reading, independently from the time.

	Change frequency	Monitoring period
Amazon – Home page	∼ Every reading	9/19/05–9/1/06
Ansa – Home page	Every 4–6 hours	9/19/05–9/1/06
Ansa – Rss sport	Every 4–6 hours	9/19/05–9/1/06
ASF France – Home page	Weekly	9/19/05–9/1/06
ASF France – Traffic page	Less than weekly	9/19/05–9/1/06
Cnn – Business	Every 4–6 hours	9/19/05–9/1/06
Cnn – Home page	Every 4–6 hours	9/19/05–9/1/06
Cnn – Weather	Daily	9/19/05–9/1/06
Java – Top 25 bugs	Less than weekly	12/1/05–9/1/06
Repubblica – Home page	Every 4–6 hours	9/19/05–9/1/06
Repubblica – Tech. and sci.	Every 2–3 days	9/19/05–9/1/06
The Server Side – Home page	Every 2–3 days	12/1/05–9/1/06
The Server Side – Tech talks	Weekly	12/1/05–9/1/06
Univ. of Trieste – Home page	Weekly	9/19/05–9/1/06
Wikipedia – Random page	Every reading	9/19/05–9/1/06

4.4 Effectiveness without retuning

We aimed to analyze the performance of the proposed approach on the short term, i.e., using a subset of the archive corresponding to a *short* temporal interval, i.e., an interval in which most of the web resources that we observed do not exhibit significant legitimate changes.

To this end, we used the full dataset—which corresponds to about six months (about 650 readings for each resource) of observations—and we manually switched off the retuning feature of the considered aggregator. In such condition, we measured the effectiveness indexes at different time points: measures gathered at the end of the first month actually corresponds to the short term effectiveness.

Table 4.2 shows FPR and FNR that we obtained in the following way. For each resource, we first built a tuning sequence composed by the first $l = 60$ readings of the corresponding S_N and we performed the learning phase on it; then, we built a testing sequence composed by the following 112 readings of the corresponding S_N and by the 100 positive readings of S_P. Note that the length of the tuning sequence—i.e., 60 readings—corresponds to 15 days of observation in our setting, and hence that would be the duration of the learning phase in a real deployment scenario; 112 readings corresponds to 28 days. Results of Table 4.2 show, as first finding, that FNR is 0 for all resources: Goldrake detected *all* the defacements included in the attack set. That is, even with a single tuning operation, Goldrake is able to detect all defacements that we considered.

On the other hand, it can be seen that even in the short period, there is some resource for with FPR is not satisfactory. Not surprisingly, we found that this is true in particular for pages which exhibit high change frequencies. This finding is confirmed below, where we present results collected at time points corresponding to the 1st month, 2nd month, and so on.

Table 4.3 shows FPR obtained in the following way. For each resource, we built a tuning sequence composed by the first $l = 60$ readings of the corresponding S_N and we performed the learning phase on it; then, we built a testing sequence composed by all the following readings of the corresponding S_N and by the 100 positive readings of S_P. Columns 2 to 5 show FPR computed at the end of each month (we considered month of 28 days, which corresponds to 112 readings), starting from the beginning of the test—i.e., FPR computed respectively on a dataset composed by 112, 224, 336 and 448 negative readings. The last column show FPR computed for the whole testing sequence, i.e., about 600 readings. Note that FNR is not shown: since the retuning feature is switched off—and hence the profile is the same as before—and the positive portion of the testing sequence is the same, the value of FNR is obviously the same as before.

Concerning false positives, these results show that FPR becomes definitely unsatisfactory after the first month, for almost all the resources. In other words, a short period strategy which does not involve some kind of retuning is not a feasible solution in the scenario of web defacement detection, where the monitored system—i.e., the web page—changes in an unpredictable way as time goes by. Eventually, the accumulation of changes makes that profile no longer useful.

	FPR (%)	FNR (%)
Amazon – Home	1.7	0
Ansa – Home	5.7	0
Ansa – Rss sport	0	0
ASF France – Home	0	0
ASF France – Traffic	0	0
Cnn – Business	0	0
Cnn – Home	0	0
Cnn – Weather	0	0
Java – Top 25 bugs	0	0
Repubblica – Home	37.5	0
Repubblica – Tech. & sci.	0	0
The Server Side – Home	20.3	0
The Server Side – Techtalks	18.0	0
Univ. of Trieste – Home	0	0
Wikipedia – Rand. page	0	0
Mean	5.5	0

Table 4.2: FPR and FNR for 15 resources computed on a dataset of about one month, using the TooManyFiringCategories aggregator (no retuning, tuning sequence length of 60 readings). Values are expressed in terms of percentage.

	After 1st month	After 2nd month	After 3rd month	After 4th month	Total (at the end)
Amazon – Home	1.7	41.0	60.7	70.1	75.3
Ansa – Home	5.7	19.6	27.4	45.5	52.1
Ansa – Rss sport	0	0	0	0	0
ASF France – Home	0	45.7	63.8	72.8	79.2
ASF France – Traffic	0	0	0	0	0
Cnn – Business	0	2.4	34.9	51.2	56.7
Cnn – Home	0	18.5	45.7	59.3	64.1
Cnn – Weather	0	18.5	45.7	59.3	64.1
Java – Top 25 bugs	0	0	na	na	0
Repubblica – Home	37.5	68.8	79.2	84.4	91.3
Repubblica – Tech. & sci.	0	27.5	51.7	63.8	68.9
The Server Side – Home	20.3	50.6	na	na	65.4
The Server Side – Techt.	18.0	59.0	na	na	75.3
Univ. of Trieste – Home	0	3.4	32.2	47.6	51.6
Wikipedia – Rand. page	0	2.8	3.4	4.0	4.6
Mean	5.5	23.9	37.1	46.5	49.9

Table 4.3: FPR for each resource computed at different time points, using the TooManyFiringCategories aggregator (no retuning, tuning sequence length of 60 readings). We considered months of 28 days, 112 readings per month. Values are expressed in terms of percentage. For some resources, results after the 3rd and 4th months are not available, because the observations for those resources lasted for less than three months (see Table 4.1).

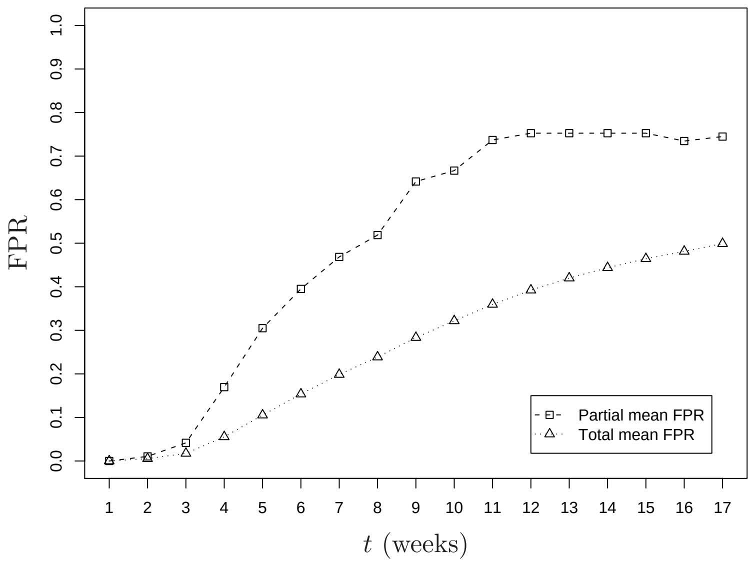


Figure 4.2: Mean False Positive Rate computed at different time points during the test (no retuning, tuning sequence length of 60 readings). Partial mean FPR for a given week is computed with the results of that week. Total mean FPR for a given week is computed with all the results obtained from the beginning of the test to the end of that given week. One week corresponds to 28 readings.

Table 4.3 shows also that it is not possible to predict how long the profile will remain useful, as this depends on the specific resource and may vary significantly.

These early findings constitute the reason for which we enabled Goldrake to update the profile of a resource, through the execution of the tuning procedure also during the normal monitoring activity. The way in which the tool decides how and when to run a tuning procedure is defined by the retuning policies, which are described in Section 3.3.3. Results obtained with such feature enabled are presented in the next section.

4.5 Effectiveness with retuning

In this section we present the results that we obtained experimenting with the automatic retuning feature enabled. In particular, we applied all the 4 retuning policies described in Section 3.3.3 to the TooManyFiringCategories aggregator and we measured FPR and FNR. Note that, since the tuning procedure is applied multiple times, in this experiments we evaluated FNR by injecting our attack set after *each* retuning. It follows that FNR will be evaluated as the ratio between the number of missed defacements and the number of injected defacements. For example, the number of injected defacements in an experiment in which the tuning procedure has been executed twice will be twice the size of the attack set. The evaluation of FPR may instead be done with the same procedure applied in the previous experiments without retuning.

	FPR (%)	FPF (f.a./weeks)	FNR (%)
FixedInterval ($n = 1$)	0.9	0.2	0
FixedInterval ($n = 15$)	3.8	1.0	0
FixedInterval ($n = 30$)	6.7	1.8	0
FixedInterval ($n = 45$)	9.7	2.6	0
UponFalsePositives	1.3	0.3	0
Mixed ($n = 15$)	1.4	0.4	0
Mixed ($n = 30$)	1.6	0.4	0
Mixed ($n = 45$)	1.7	0.4	0
SelfVerifying ($n = 30$)	5.2	1.4	0

Table 4.4: Mean FPR, FPF and FNR for nine retuning policies, using the TooManyFiringCategories aggregator (tuning sequence length $l = 60$ readings).

Table 4.4 shows results that we obtained experimenting with different values for the retuning interval n (see Section 3.3.3), when applicable; this figures are obtained using a tuning sequence length of $l = 60$ readings. In other words, Goldrake maintains a sliding window of the last most recent readings and use it when a retuning is performed. Results of Table 4.4 are averaged across all the 15 resources and are also cast in terms of FPF.

Goldrake still exhibits FNR equal to 0% for each resource, that is, it retains the ability of detecting every defacement included in the attack set. The key result is that FPR becomes tolerable thanks to retuning. Tuning procedures involving a human operator achieve FPR in the range 1.4–1.7%, as opposed to 49.9% without retuning. Even fully automatic procedures perform well, a retuning performed every 15 readings (about 4 days), for example, leads to FPR = 3.8%. An automatic retuning performed at every reading (FixedInterval with $n = 1$) leads to FPR equal to 0.9%, which is slightly better than the one obtained with human-triggered policies. Note that, when expressed in terms of FPF, these results show that it is possible to obtain less than 1 false positive each month, for each page. This finding suggests that the proposed approach may be a feasible solution for large-scale monitoring, and hence deserves further attention.

Tables 4.5 and 4.6 show the same experiment results separately for each resource (FPR data only). It can be seen that there are no intractable resources, i.e., resources for which FPR is unsatisfactory. The best retuning policy—FixedInterval ($n = 1$)—leads to FPR never greater than 2%.

4.6 Impact of tuning sequence length

The results of the previous section have been obtained with a tuning sequence composed of 60 readings. We repeated the previous experiments with other tuning sequence lengths: 30 and 90. Table 4.7 shows the mean of False Positive Rates for four selected policies with three values for the tuning sequence length l, again for the sole TooMany-

	FixedInterval				UponFalse-Positives
n	1	15	30	45	
Amazon – Home	0	0	2.2	4.4	1.5
Ansa – Home	0.4	0.7	2.9	3.9	1.8
Ansa – Rss sport	0.6	2.6	2.6	1.7	0
ASF France – Home	1.7	4.4	9.2	14.5	0.7
ASF France – Traffic	0	0	0	0	0
Cnn – Business	1.7	6.4	9.8	15.7	2.6
Cnn – Home	1.5	5.5	6.8	10.7	2.6
Cnn – Weather	0.9	4.4	5.2	5.7	1.7
Java – Top 25 bugs	0.7	4.3	9.4	4.3	0
Rep. – Home	0.7	5.2	8.3	19.5	1.3
Rep. – Tech. & sci.	2	9.6	17.9	24.3	0.9
The Serv. S. – Home	1.7	2	3	3	2.3
The Serv. S. – Tech talks	1	3.7	7.7	13.7	0.7
Univ. of Trieste – Home	1.5	7.7	15.4	23.3	2.8
Wikipedia – Rand. page	0	0.2	0.6	1.3	1.3
Mean	0.9	3.8	6.7	9.7	1.3

Table 4.5: FPR of Table 4.4 expressed separately for each resource (FixedInterval and UponFalsePositives retuning policies).

	Mixed			Self
n	15	30	45	30
Amazon – Home	0	0.6	0.9	0
Ansa – Home	0.6	0.7	0.7	0.4
Ansa – Rss sport	0.7	0.2	0.2	2.4
ASF France – Home	1.7	1.8	1.7	9.2
ASF France – Traffic	0	0	0	0
Cnn – Business	2.4	2.8	2.4	9.6
Cnn – Home	2.2	2.4	2.4	3.9
Cnn – Weather	1.7	1.7	1.8	5.2
Java – Top 25 bugs	0.7	0.7	0.7	9.4
Rep. – Home	2.6	2.8	2.9	4.6
Rep. – Tech. & sci.	3.1	3.1	3.1	16.4
The Serv. S. – Home	1.7	2	1.7	1.3
The Serv. S. – Tech talks	1.3	1.7	1.7	7.7
Univ. of Trieste – Home	2.4	3.1	3.7	7.2
Wikipedia – Rand. page	0.2	0.6	1.3	0.2
Mean	1.4	1.6	1.7	5.2

Table 4.6: FPR of Table 4.4 expressed separately for each resource (Mixed and Self retuning policies).

	$l = 30$	$l = 60$	$l = 90$
FixedInterval ($n = 30$)	12.8	6.7	5.7
UponFalsePositives	2.2	1.3	0.9
Mixed ($n = 30$)	2.7	1.6	1.3
SelfVerifying ($n = 30$)	9.3	5.2	5.0

Table 4.7: Mean FPR, expressed in percentage, for four selected policies with different values for the tuning sequence length l, using the TooManyFiringCategories aggregator.

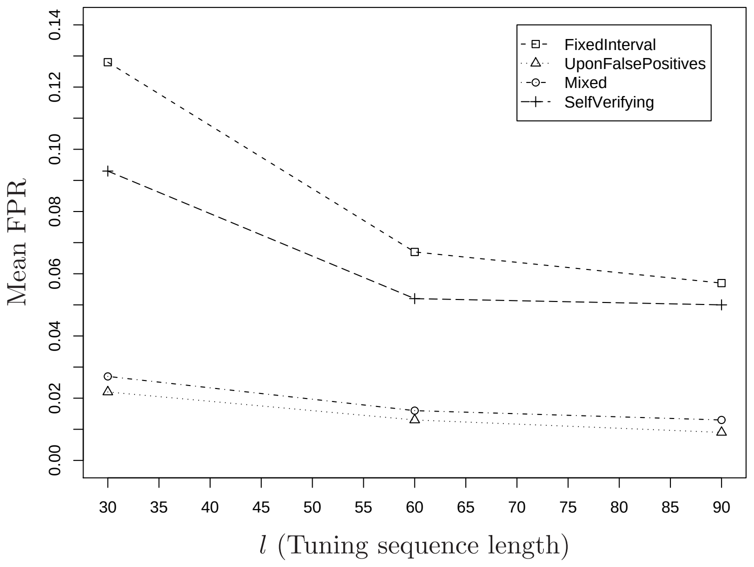


Figure 4.3: Mean False Positive Rate vs. Tuning sequence length l for four selected policies, using the TooManyFiringCategories aggregator.

FiringCategories aggregator. These results are also shown in Figure 4.3 in a graphical way.

Interestingly, results show that the performance of human-triggered retuning policies is largely independent of the tuning sequence length. In contrast, the performance of the automatic retuning policy improves when the tuning sequence length increases. This observation may be exploited in scenarios when the involvement of a human operator in retuning is undesirable (see also Section 4.9).

4.7 "Fast start" working mode

We found that increasing the tuning sequence length may reduce FPR, in particular, when the retuning policy is fully automatic (see Table 4.7). This option, however, has the drawback of increasing the time required for bootstrapping the tool: the learning phase will last longer thus the monitoring phase will start later. For example, in our

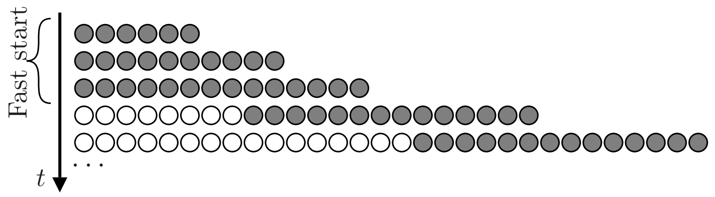


Figure 4.4: An example of the fast start strategy. Filled circles represent readings that are contained in the tuning sequence. Each line of circles corresponds to a time point in which a retuning is performed; readings downloaded between two retunings are evaluated but they are not shown in the figure. In this case $l_i = 6$, $\Delta l = 4$ and $l_f = 14$.

setting, with a tuning sequence length of $l = 90$ readings the learning phase will last for 23 days, whereas with $l = 60$ readings the learning phase will last for 15 days. In this section we describe and evaluate a working mode, that we call *fast start*, aimed at shortening the bootstrap time.

In this working mode the monitoring phase starts after Goldrake has collected a short tuning sequence, whose length is then increased over time until reaching its target value. We remark, however, that a suitable historical archive of the resource to be monitored could often be already available. In such scenarios the learning phase may be performed by simply "replaying" the content of the archive, which would lead to an immediate start of the detecting phase.

Goldrake executes the first tuning procedure on a sequence of l_i readings and then starts monitoring. A retuning is performed each Δl readings, on a tuning sequence obtained by appending the last Δl readings to the previous tuning sequence. When the tuning sequence length reaches its target value of l_f readings, the fast start terminates and Goldrake begins working as usual, i.e., with $l = l_f$. Figure 4.4 shows an example of the fast start strategy.

We experimented with three selected retuning policies: FixedInterval ($n = 30$), UponFalsePositives, Mixed ($n = 30$). We ran the three policies in fast start mode, with $l_i = 15, \Delta l = 15$ and $l_f = 90$. With this setting, the learning phase lasted for about 4 days and the fast start phase lasted for about 19 days (one increment every 4 days for 5 times). As a comparison we ran the same policies in normal mode, i.e., without fast start, with the two extreme values for the tuning sequence length: first with l_i readings, then with l_f readings. In the former case the tool begins the monitoring phase with the same initial delay (about 4 days), but has less information available for building the profile. In the latter case the tool has the same amount of information for building the profile but starts the monitoring phase after a much longer initial delay (about 19 days). The results are given in Table 4.8.

It can be seen that the fast start mode is very effective. For the same initial delay, it provides a remarkable FPR reduction—e.g., 12.2% vs. 6.6% for FixedInterval. For the same length of the tuning phase after the initial transitory, it provides nearly the same FPR—e.g., 6.6% vs. 5.7% for FixedInterval. Indeed, analysis of the raw data shows that the slightly larger FPR is caused by the initial inaccuracies in the profile. Near the end

	Fast start mode	Normal mode	
	$n = 30$ $l_i = 15$ $l_f = 90$	$n = 15$ $l = 15$	$n = 30$ $l = 90$
FixedInterval	6.6	12.2	5.7
UponFalsePositives	1.4	3.3	0.9
Mixed	1.7	4.7	1.3

Table 4.8: Effect of the Fast Start working mode, using the TooManyFiringCategories aggregator. Mean FPR for three selected policies with different values for the tuning sequence length, using. The parameter n is used where applicable, i.e., in policies Mixed and FixedInterval.

of the test the differences between fast start and normal mode vanish.

4.8 Effectiveness of sensor categories

In order to gain insights about the behavior of individual sensors, we isolated the effectiveness results as obtained before the merging of the TooManyFiringCategories aggregator. To this end, we performed the same experiment as above and we counted false positives and false negatives raised by individual sensors.

Recall that sensors are functional blocks and hence they do not hold any state information, nor profiles, which are indeed part of the aggregator (see Section 3.3.1). However, as explained in Section 3.3.2, the profile of TooManyFiringSensors and TooManyFiringCategories aggregator is actually "partitioned" in pieces corresponding to individual sensors; thereby, for those aggregators, we can say that a sensor fires whenever the corresponding piece of v is not consistent with the corresponding piece of the profile.

Table 4.9 summarizes the FPR and FNR that one would obtain with each sensor taken in isolation. These results are obtained experimenting with 4 retuning policies operating with a learning sequence of $l = 60$ readings on the whole sequence of about 650 negatives and 100 positives. For ease of discussion we report only the average values for each category. For example, tree sensors exhibit a mean FPR of 12.6% and a mean FNR of 4.6%. Note that signature sensors exhibit, as expected, the same FPR and FNR for every retuning policy, because they do not build any profile of the resource.

These results suggest two important observations. First, there is no single category providing good performance on both FPR *and* FNR. Each category provides satisfactory performance only on one of the two indexes and tree sensors appear to provide the best trade-off in this respect. Second, signature sensors are by far the least effective ones in detecting defacements. In other words, looking for signs of a defacement is much less effective than detecting deviations from a profile. This rather surprising result implies that, for the defacement detection problem, an anomaly-based approach may be more sensitive than a misuse-based one (by this we mean an approach that looks for traces of known attacks) [16].

	FixedInterval ($n = 30$)		UponFalse Positives		Mixed ($n = 30$)		Self Verifying ($n = 30$)	
	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR
Cardinality	5.6	19.8	9.1	19.7	5.2	19.6	4.6	18.9
RelativeFreq.	12.4	0.0	17.1		11.4	0	9.8	0
HashedItemsC.	15.1	0	25.0	0	11.1	0	10.6	0
Signature	0	66.2	0	66.2	0	66.2	0	66.2
HashedTree	12.6	4.6	10.7	7.1	8.4	4.3	7.9	4.3

Table 4.9: FPR and FNR averaged among the sensors of the five categories and obtained with four different retuning policies and a tuning sequence length of 60 readings.

4.9 Scalability

In this section we attempt to assess the potential scalability of our proposal. Clearly, our arguments in this respect need to be validated by a real deployment of the proposed service, involving thousands of resources and organizations for several months. Such a deployment, which we have not been able to perform so far, is beyond the scope of this work.

4.9.1 Number of resources

In this section we attempt to estimate the maximum number of resources that can be monitored, say N_{MAX}. This number depends mainly on the retuning policy used, as follows.

In case all the resources are associated with a fully automatic retuning policy, then it seems reasonable to claim that N_{MAX} is limited only by the hardware resources available. The reason is because, in principle, human operators at Goldrake need not be involved in the management of individual resources. Addition of a new monitored resource may be done by the administrator of that resource, with the help of a dedicated web application, for example. Alarm forwarding may also be done in a fully automatic way. Moreover, due to the intrinsic parallelism of the process, N_{MAX} may be increased by simply adding more hardware—the monitoring of distinct resources may be done by independent jobs, without any synchronization nor communication between them. It is thus possible to take advantage of this job-level parallelism with techniques similar to those commonly used for achieving horizontal scalability in web sites designed to accommodate large numbers of clients, for example.

Concerning resources associated with a human-triggered policy, the scalability bottleneck is given by false alarms. Let A_{FPF} denote the False Positive Frequency exhibited by Goldrake for a single resource, let T_{FPF} denote the maximum False Positive Frequency that can be tolerated by a single human operator and let k denote the number

T_{FPF} f.a./day	T_{FPF} f.a./week	N_{MAX}
1	7	35
2	14	70
4	28	140
20	140	700

Table 4.10: Maximum number of monitored resources N_{MAX} for different FPF tolerated by a single ($k = 1$) human operator T_{FPF}.

of operators at Goldrake involved in the handling of human-triggered retuning policies. Assuming that all resources exhibit the same A_{FPF}, we obtain the following estimate for N_{MAX}:

$$N_{\text{MAX}} \cong k \frac{T_{\text{FPF}}}{A_{\text{FPF}}} \tag{4.1}$$

According to our findings in the previous sections, we can reasonably assume for A_{FPF} a value in the order of 0.2 false positives per week, on each resource (policies UponFalsePositives with a tuning sequence of $l = 90$ readings and FixedInterval ($n = 1$) with $l = 60$). Finding a suitable value for T_{FPF}, on the other hand, is clearly more difficult and hard to generalize. Table 4.10 shows the values of N_{MAX} resulting from one single operator ($k = 1$) and T_{FPF} set to $1, 2, 4, 20$ false positives per day. For example, if the operator may tolerate 4 false positives per day, then he may monitor 140 resources. From a different point of view, if one monitors 140 resources, then one may expect that the operator be involved only 4 times a day for coping with false positives. Clearly, with $k > 1$ operators available, the corresponding estimates for N_{MAX} will grow linearly with k.

In summary, the approach appears to be: linearly scalable for resources whose retuning is fully automatic; capable of monitoring several hundreds of resources per operator, for resources whose retuning is human-triggered.

We remark that an actual deployment of the service would have several options available, for example:

1. offer only fully automatic retuning;

2. offer both automatic retuning and human-triggered retuning, by pricing the two options differently—e.g., "silver level" vs. "gold level" surveillance;

3. offer only fully automatic retuning, with pricing differentiated based on the retuning frequency (recall that by increasing this frequency the FPR performance of automatic retuning becomes equivalent to that of human-triggered retuning, as shown in Table 4.4).

N	m_l mins:secs
35	0:46
70	1:32
140	3:03
700	15:17

Table 4.11: Minimum monitoring interval m_l for N resources monitored simultaneously on our test platform (see text).

4.9.2 Monitoring interval

An estimate of the lower bound for the monitoring interval, say m_l, can be obtained by reasoning on the time it takes for downloading a reading, say t_d, the time it takes for evaluating that reading with all sensors, say t_e, and the time it takes for performing a retuning procedure, say t_r. A very conservative estimate can be obtained by assuming that there is no overlap at all among downloading, evaluation and retuning.

We can devise two extreme cases. The worst case is when a retuning is performed at every reading for each of the N monitored resources. In this case it will be:

$$m_l > (t_r + t_d + t_e) \cdot N \tag{4.2}$$

The best case is when no retuning is performed, in which case it will be:

$$m_l > (t_d + t_e) \cdot N \tag{4.3}$$

In practice, the minimum allowable m_l will be somewhere in between the two bounds above depending on the retuning policies used for the resources. The average values for t_d, t_e and t_r that we found in our experiments with the TooManyFiringCategories aggregator (which is the more computationally demanding, among the 3 presented in Section 3.3.2) are, respectively, 1300msecs, 100μsecs and 10msecs: hence, m_l is influenced mainly by the download time t_d. These values were obtained with a dual AMD Opteron 64 with 8GB RAM running a Sun JVM 1.5 on a Linux OS. The resulting lowest bound for m_l that we can achieve in our current setting, considering the case in which a retuning is performed at every reading for each resource (i.e., eqn. (4.2)), is shown in Table 4.11. For example, if Goldrake monitors 140 resources, then an attack may be detected in approximately 3 minutes. From a different point of view, if one requires a detection time in the order of 3 minutes, then no more than 140 resources should be monitored (with one single computing node at Goldrake).

We remark again that these figures have to be interpreted along with the scalability analysis in Section 4.9.1: management of different resources are independent jobs, hence the monitoring process is linearly scalable in the number of computing nodes (except for the FPR issues related to human-triggered retuning policies, which are orthogonal to this analysis).

4.10 Temporal correlation of alerts

In all our tests we used a monitoring interval of 6 hours, thus collecting readings four times per day. This means that Goldrake is able to detect a defacement within 6 hours after its occurring. A prompter detection may be obtained by simply decreasing the monitoring interval. Although the lower bound on the monitoring interval ultimately depends on the available resources (see also Section 4.9.2), decreasing the length of the monitoring interval might increase the number of alerts submitted to the administrator. In this section we elaborate on this issue.

Consider a resource R at which a substantial but legitimate change is applied at instant t_1 and suppose that, due to this change, the profile of R is no longer adequate. It follows that all readings of R analyzed between t_1 and the next retuning, say at instant t_r, will provoke false positives.

With the FixedInterval retuning policy, the number of false positives before the next retuning will be:

$$\frac{t_r - t_1}{m} \tag{4.4}$$

where m denotes the length of the monitoring interval. If m decreases, the number of false positives will increase. The rate of false positives will not change, since with a shorter m we have both more false positives and total readings. This effect is shown in Figure 4.5 (policies FixedInterval (A) and FixedInterval (B)). The key observation, however, is that the consecutive false alarms are correlated in the sense that they are all expression of the same problem.

With a human-triggered retuning policy the generation of false positives is quite different. Since we execute a retuning after the first false positive, the number of false positives is always 1, *independently* from the length of the monitoring interval (policies UponFalsePositives (A) and UponFalsePositives (B) in Figure 4.5). In summary, with an automatic retuning policy decreasing the monitoring interval will lead to a linear increase of FPF but the alarms will be correlated with each other. With a human-triggered retuning policy, instead, the monitoring interval can be freely decreased without suffering an FPF increase.

In this reasoning we have implicitly assumed that a single false alarm may suffice for updating the profile adequately. We verified that this hypothesis indeed holds in practice. We evaluated the performance for FixedInterval ($n = 30$) and UponFalsePositives by counting each set of *consecutive* false alarms as a single false alarm. Table 4.12 compares the results to those previously found for the corresponding policies when taking into account all false alarms separately. It can be seen that UponFalsePositives evaluated without consecutive false alarms performs very similarly to UponFalsePositives. In other words, with the latter, when Goldrake raises a false alarm it usually does not raise any further alarms in the readings that immediately follow. These observations appear to confirm that a single false alarm may suffice for updating the profile adequately.

Another interesting finding is that results for the FixedInterval ($n = 30$) retuning policy, evaluated without consecutive false alarms, are much better than those previously found for FixedInterval ($n = 30$). This fact demonstrates that, in our setting, most of

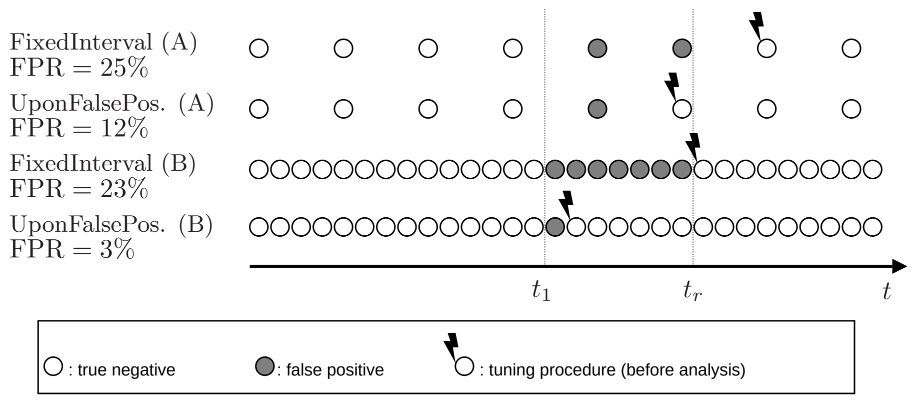


Figure 4.5: The effect of decreasing the monitoring interval m (i.e., in the figure, the distance between two circles). Dotted line in t_1 represents a time point in which a legitimate, but substantial, change is applied to the resource. Due to such change, the profile of the resource is no longer adequate hence the tool will raise false positives until the next retuning. With a human triggered policy, the tuning procedure is executed as soon as a false alarm is produced. With an automatic retuning policy, the tuning procedure is executed at fixed time point t_r.

	FPR (%)	FPF (f.a./weeks)
FixedInterval ($n = 30$)	6.7	1.8
FixedInterval ($n = 30$) w/o consecutives	1.5	0.4
UponFalsePositives	1.3	0.3
UponFalsePositives w/o consecutives	1.1	0.3

Table 4.12: Mean False Positive Rate and mean False Positive Frequency for two retuning policies compared to corresponding measures where consecutive false alarms are considered as a single false alarm. FPR and FNR are percentage; FPF is expressed in terms of false alarms per week. Results have been obtained with a tuning sequence length of $l = 60$ readings, with the TooManyFiringCategories aggregator.

false alarms raised by FixedInterval were actually consecutive alarms. This observation could be exploited in practice for simplifying the job of the administrator. The alarm sent to the administrator of the monitored site could be augmented with an indication telling whether the alarm is correlated with the alarm sent at the previous step—i.e., it is still the same problem.

4.11 Discussion

Experiments show that Goldrake is always able to detect all the defacements contained in our attack set, using the TooManyFiringCategories aggregator: FNR is 0% for all resources, in every operational condition that we analyzed. Obviously, this result does not imply that Goldrake will be able to detect any possible unauthorized modification in any resource. Yet, it seems reasonable to claim that Goldrake may be very effective at detecting "major" defacements. From this point of view, it is perhaps surprising that looking for signs of a defacement is not an effective approach (see Table 4.9 and Section 4.8).

It would certainly be useful if we were able to quantify somehow the opportunities for attacks that may remain hidden within the profile. This is a very challenging problem beyond the scope of this study, however. The problem is essentially the same as quantifying the opportunities of successful attacks that would be left by a given intrusion detection system or a given anti-virus defense. We can only tell, from a qualitative point of view, that increasing the number of sensors may certainly be useful for restricting the space of possible undetected attacks.

Our current set of sensors is certainly not exhaustive and an area where Goldrake could be improved is the analysis of textual content. None of our sensors attempts to build a profile of common sentences or of the language used, for example. Exploiting this form of knowledge is not as easy as it could appear, however, due to the difficulty of obtaining meaningful alerts in the presence of a highly dynamic dictionary. In our early attempts we did not manage to obtain a sensor more useful than the one looking for missing recurrent words in a resource (see Section 3.3.2). This area certainly deserves further thought, however.

Apart from the ability to detect attacks, the other major issue is false positives, that is, how to cope with (legitimate) dynamic content while keeping unnecessary alerts to a minimum. The crucial result of our experiments is that it is indeed possible to obtain values for FPR sufficiently low to appear acceptable in a wide variety of settings. In this respect, a key contribution consists in the retuning policies. Even with a fully automatic retuning performed every $n = 30$ readings, FPR remains moderate: 6.7%, which corresponds to approximately 8 false alarm per month per resource. By retuning more frequently FPR improves further: 3.8% with $n = 15$ and 0.9% when retuning at every reading ($n = 1$). Interestingly, these figures are essentially equivalent to those of human-triggered retuning. From a different point of view, the job of the operator involved in retuning may be to a large extent automated. Obviously, we do not claim that site administrators may simply forget about defacements thanks to Goldrake. No security tool may replace human operators completely, amongst other things because no detection system may catch all possible attacks.

It would be also very useful if we could quantify somehow the potential for false positives based on some index about the accuracy of the resource profile built during the learning phase. We purposefully did not address the problem of assessing the accuracy of a profile, though, because any such analysis would require some form of a priori knowledge about the monitored resource that instead we strive to avoid. For example, if an image appears in every reading of the learning phase, there is simply no way to tell whether that image is a cornerstone of the profile or it will instead disappear immediately after the end of the learning phase. One could perhaps obtain more meaningful indications much after the learning phase, based on data collected during a "long" monitoring phase, but quantifying the potential for false positives at that point would not be very useful. Stated differently, our problem cannot be interpreted merely as the construction of an accurate profile, because it consists instead of a trade-off between accuracy and speed of construction. Rather than attempting to quantify accuracy of a profile, thus, we decided to revert to the common practice in anomaly-based detection systems [54, 49, 66, 77, 41]: validate our proposal experimentally by measuring FPR (and FNR) on several different resources that we observed for a long period and that vary broadly in content, frequency and impact of changes. The same remarks can be made with respect to the training set size, that is often set to a fixed, domain knowledge-based value in order to cope with the practical impossibility of automatically determining when enough training data has been collected [54].

Our sensors and aggregator exploit a form of domain-specific knowledge (for example, by grouping sensors in categories) and it may be interesting to note that we tune resource-specific parameters in a very simple way (for example, in the TooManyFiringCategories aggregator, the threshold is either a constant value or a predefined function of mean and standard deviation of the indicator). Despite its simplicity, this approach allowed us to obtain a very low FPR. A similar remark can be made with respect to the effectiveness of our retuning policies, which provide very good FPR despite their simplicity—we rebuild profiles either at regular intervals or when a false alarm occurs. However, we did investigate whether more sophisticated techniques that have been applied to Intrusion Detection Systems would improve our results significantly. Results are presented in Chapter 7.

Although our approach does not require any knowledge about the monitored resource, it does allow exploiting such a knowledge when available. For example, in many sites external links are not supposed to contain sex-related words. One could add a sensor that fires when one or more of such undesired words are present in an external link. The early discussions about this work were indeed triggered by a successful attack in our administrative region that left the appearance of the site unmodified and only redirected external links to pornographic sites. Recently, some U.S. Government web sites were successfully attacked in a similar way [48].

New aggregators can be designed in the same way. For example, the firing of a sensor detecting a change in the URL associated with a specified HTML form (perhaps the one where the credentials of an Internet banking user have to be inserted) could be made a sufficient condition for raising an alert, irrespective of the output of the other sensors. Such forms of personalization could be part of different levels of service offered by the organization running Goldrake. We remark again that we paid special attention to make Goldrake modular and extensible. New sensors and new aggregators can be built easily, as Java classes that implement a certain interface. Each resource may be associated with its own set of sensors and with its own aggregator, taken from the available building blocks. A given resource may also be monitored in several different ways, perhaps using different sensors and/or aggregators, which may be useful for attaching a form of severity level to the alert.

Finally, we remark that the tuning sequence should clearly provide "sufficient" information about the resource, otherwise one may end up with a profile that is not meaningful. At the same time, the tuning sequence should include only genuine readings, i.e., it should be free by defacements. This is also a problem intrinsic of any anomaly-based approach: understanding and quantifying how the presence of non-genuine readings in the tuning sequence affect the detector effectiveness is an interesting issue. We analyze this problem and propose a partial solution in Chapter 5.

Chapter 5

Effects of misclassified readings in the learning set

Experimenting with Goldrake, we found that the approach we propose may indeed constitute a feasible solution for the automatic detection of web site defacement. However, similarly to other anomaly detection approaches, the effectiveness of our solution bases on the assumption that the profile is consistent with the monitored system. In particular, we assume that it is computed starting from a “good” *learning set*, i.e., a learning set that does not contain any attack.

Despite being anomaly detection widespread and the good learning set assumption equally common, the problem of assessing the effects of its negation is substantially not treated in literature. In this chapter, we focus on this topic and present results obtained experimenting on our particular instance of anomaly detection. Hence, the scope of the following analysis is clearly narrowed by the specific detection system that we consider, but we believe that our considerations may be of interest also for other forms of detection systems.

5.1 Scenario: anomaly detection and learning set corruption

Anomaly detection is a powerful and commonly used approach for constructing intrusion detection systems. With this approach the system constructs automatically a profile of the resource to be monitored, starting from a collection of data representing normal usage (the learning set). Once this profile has been established the system signals an anomaly whenever the actual observation of the resource deviates from the profile, on the assumption that any anomalies represent evidence of an attack.

A key requirement is that the learning set is indeed attack-free, otherwise the presence of attacks would be incorporated in the profile and, thus, considered as a normal status. Although this requirement is crucial for the effectiveness of anomaly detection, in practice the absence of attacks in the learning set is either taken for granted or verified “manually”. While such a pragmatic approach may be feasible in a carefully controlled environment, it clearly becomes problematic in many scenarios of practical interest. Building a large number of profiles for resources immersed in their production environment, for example, cannot be done by inspecting each learning set “manually” to make sure there were no attacks. A cross-the-fingers approach, on the other hand, can only lead to the construction of potentially unreliable profiles.

We consider the problem of *corruption* in the learning set, i.e., a learning set containing records which do not represent a “normal” condition for the observed system. We think that focusing on this fundamental issue could broaden the scope of anomaly-based detection frameworks. For example, in order to apply anomaly-based monitoring on a large scale—to hundreds or even thousands of system instances—it is necessary to build a large number of learning sets, one for each instance to be monitored (e.g., [37] monitored many web applications at the same time). It is clearly not feasible to “manually” check each learning set to make sure there are no attacks hidden in it. The fact that profiles could have to be updated periodically in order to reflect changes in legitimate usage of resources (e.g., [66]) may only exacerbate this problem.

A road map to deal with the problem of potentially corrupted learning sets involves:

1. *understanding*, i.e., evaluating quantitatively the effects of a corrupted learning set on the effectiveness of anomaly detection;

2. *detecting*, i.e., being able to discriminate between a corrupted learning set and a clean one;

3. *mitigating*, i.e., preserving an acceptable performance level of the detection system in spite of a corrupted learning set.

We focused on the first two steps.

To this end we performed a number of experiments on Goldrake, basing on collections of real data and considering three different flavors of anomaly detection—i.e., three aggregators. We first assess the effects of a corrupted learning set on performance, in terms of false positives and false negatives.

5.2 Related work

Broadly speaking, anomaly detection is an instance of *inductive learning classification* in which the goal is to build a profile able of discriminating between only two classes—i.e., normal and anomalous—using learning data corresponding to a single class—i.e., normal [38, 39, 42, 3]. In the inductive machine learning field the corruption of learning set is indicated as *noise*, which is generally subdivided in two categories: attribute noise and class noise. The former consists in records for which one or more attributes are not really representative of the corresponding class, whereas the latter concerns records of the dataset which are wrongly labeled. A comparison between the effects of the two different types of noise on classifier accuracy is presented in [82]; our work refers to a very specific instance of the inductive machine learning problem and considers only class noise.

Concerning class noise, there is a substantial amount of work proposing solutions for identifying and then removing the corrupting (mislabeled) records. More in general, this issue can be addressed with *outlier detection* techniques [27]. We have not investigated whether such techniques can be applied to our framework, that is characterized by a small learning set—usually a few tens of records. In this work we are merely concerned with the problem of detecting whether the learning set is indeed clean or contains some amount of class noise.

An important method for finding mislabeled records in the learning set is given in [8]. The idea consists in building a set of filtering classifiers from only part of the learning set and then testing whether data in the remaining part also belong to the profile. The learning set is partitioned in a number of subsets and, for each subset, a filtering classifier is trained using the remaining part of the learning set. Each record of the learning set is then input to each of the filtering classifiers. The cited paper proposes and evaluates several criteria for merging the labels generated by the filtering classifiers. The method should be able to identify outliers regardless of the specific classifier being used, hence, regardless of the chosen model for the data. A very similar approach, specifically tailored to large datasets, is proposed in [82]. Our work differs from these proposals in that we do not partition the learning set in smaller sets. This can be an advantage in the cases—our test scenario is indeed one of them—where learning sets may be very small and thus a further division will lead to an ineffective classifier: such cases are considered by Forman and Cohen [19] in their comparative study about machine learning applied to small learning set.

Concerning specifically anomaly detection, we are not aware of works covering both the understanding and detecting phases of the problem of corrupted learning sets. We are only aware of a few published experiments about the effects of a corrupted learning set—i.e., only the understanding phase. Hu et al. [29] consider an anomaly-based host intrusion detection system and compare the performance of Robust Support Vector Machines (RSVMs), conventional Support Vector Machines and nearest neighbor classifiers using 1998 DARPA BSM data set. Besides experimenting with clean learning sets, they consider also artificially corrupted learning datasets and find that RSVM are more robust to noise.

A similar analysis is proposed by Mahoney and Chan [46]. The authors present an IDS which detects anomalies in packet header fields. In addition to the normal effectiveness evaluation (with 1999 DARPA dataset), they experiment also with smaller and corrupted learning sets. They motivate this choice based on the practical difficulty in obtaining attack-free learning sets. The authors test their tool in a real environment and retune the tool every day based on the data collected on the previous day. A significant loss in detection rate is highlighted, but the relation between loss and corruption is not analyzed because the corruption level is not measured accurately.

A radically different approach to the problem of corrupted learning sets is that of *unsupervised anomaly detection*. With these techniques learning set records do not need to be labeled as clean or attack-related and the detection method itself is intrinsically robust to a certain amount of noise. Along this line, Laskov et al. [40] present a network IDS based on a formulation of a one-class Support Vector Machine (SVM) [73], whereas Wang and Stolfo [74] present a network IDS where anomalies with respect to the profile are based on the the Mahalanobis distance. These techniques usually work on learning datasets much larger than ours, which often consists of only a few tens of records.

5.3 Experimental evaluation

5.3.1 Dataset

In order to perform our experiments, we used an extended version of the dataset collected in order to test Goldrake (see Section 4.2). It has composed observing 15 web pages for about one year, collecting a reading for each page every 6 hours, thus totaling about 1350 readings for almost each web page. These readings compose the *negatives sequences*—one negative sequence S_N for each page: we visually inspected them in order to confirm the assumption that they are all genuine, that is, none of them was a defacement.

We also built a single *positive sequence* S_P composed by 100 readings extracted from a publicly available defacement archive. Defacements composing S_P were not related with any of the 15 resources that we observed—as pointed out above none of these resources was defaced during our monitoring period. The next section explains how we used S_P readings in order to simulate attacks to the monitored resources.

5.3.2 Methodology

We wanted to gain insight about how a given aggregator copes with a corrupted learning sequence S_{learning}, i.e., when S_{learning} contains readings that must be classified as anomalous. We considered different *corruption rates* r, i.e., different fractions of S_{learning} composed by positive readings. We measured the aggregator effectiveness, in terms of false positive rate (FPR), false negative rate (FNR) and area under the ROC curve (A_{ROC}).

For each aggregator A, corruption rate r and page p:

- We constructed a learning sequence S_{learning} as follows.
  1. We extracted a sequence $S = \{i^n_0, \dots, i^n_{125}\}$ of 125 consecutive readings from the negative sequence S_N of the page p.
  2. We split S in two subsequences: S'_{learning} composed by the first $l = 50$ readings and S'_{test} composed by the last 75 readings.
  3. We constructed S_{learning} by replacing the $50 \cdot r$ final readings of S'_{learning} with a positive reading extracted from the positive sequence S_P and repeated $50 \cdot r$ times (for simulating a defacement occurred while collecting the learning data). Note that r represents the percentage of S_{learning} that is corrupted.
- We constructed a test sequence S_{test} for evaluating the profile built with the corrupted learning sequence S_{learning} as follows.

4. We inserted at the beginning the sequence S'_{test} obtained at the previous step.
5. We appended 75 different positives extracted at random from the positive sequence S_P (and including the one that corrupted S_{learning}).

In other words S_{test} is composed of 150 readings, the expected output should be negative in the first half and positive in the second half.

We repeated the above experiment several times, with $N_S = 10$ different negative sequences S at step 1 and with $N_p = 10$ different positive readings at step 3. For each tuple $\langle \text{page } p, \text{aggregator } A, \text{corruption rate } r \rangle$, thus, we performed $N_S N_p = 100$ experiments evaluating FPR, FNR and A_{ROC} in each experiment. The values in the next sections are the average values obtained across all 15 pages in our dataset.

5.3.3 Results

5.3.4 Uncorrupted learning sequence

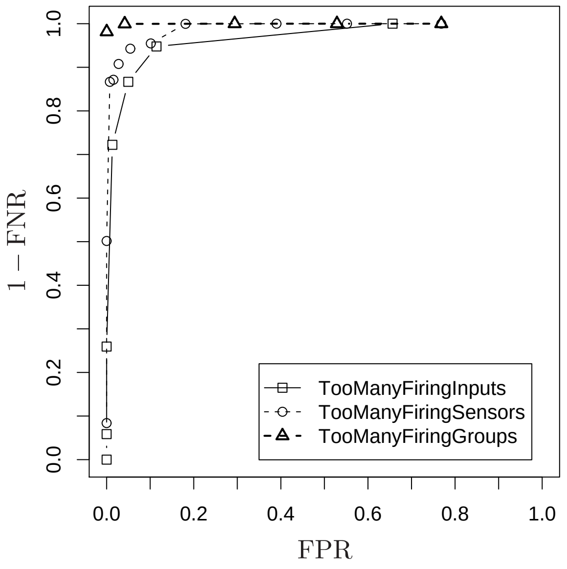
In this section, we present the results obtained for the 3 aggregators presented in Section 3.3.2 evaluated on an uncorrupted learning sequence, i.e., with $r = 0$. The values obtained for FPR, FNR and A_{ROC} in these conditions will serve as a comparison baseline.

Table 5.1 shows FPR and FNR for the 3 aggregators as a function of the threshold t. The same results are plotted in Figure 5.1 in the form of ROC curves. It is clear that TooManyFiringCategories outperforms the other aggregators. In particular, Table 5.1 confirms that, with $t = 0.9$, TooManyFiringCategories never misclassifies a negative reading as positive, while it wrongly undetects only 1.9% of positive readings (i.e., defacements). Such values can not be obtained with any t value for the two other aggregators. This results confirms the intuition that domain-specific knowledge may be very beneficial in the design of an aggregator (this is similar to, e.g., choosing values for conditional probabilities in a Bayesian network [54]).

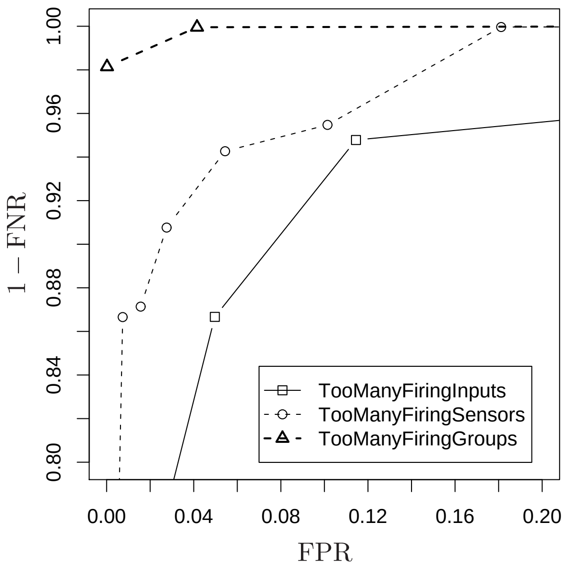
The choice of the threshold t depends on the desired trade-off between FPR and FNR. We have emphasized in bold in Table 5.1 the values corresponding to the lowest value of FPR+FNR—just one of the possible performance indexes. We observe that the 3 aggregators have different values for the respective optimal threshold t_{opt}. Assigning the same weight to false positive and false negatives may or may not be appropriate in all scenarios. For example, in the web site defacement detection scenario, one could tolerate some false positive in order to be sure of not missing any defacement. On the contrary, in the spam detection problem, one could accept some undetected spam message while not tolerating a genuine e-mail being thrown away.

Corrupted learning sequence

In this section, we present the results concerning the effectiveness of the aggregators when corrupted learning sequences are used. We experimented with the following values for the corruption rate r, including 0 (the uncorrupted sequence): $0, 0.02, 0.05, 0.1, 0.2, 0.35, 0.5, 0.75$.



(a) ROC curves



(b) ROC curves (detail)

Figure 5.1: ROC curves for the 3 aggregators obtained with $r = 0$, i.e., with uncorrupted learning sequences. The plot on the right shows the area with FPR and FNR lower than 20%.

Table 5.1: FPR and FNR for the 3 aggregators and several t values obtained with $r = 0$, i.e., with uncorrupted learning sequences, expressed in percentage. Values corresponding to $t = t_{\text{opt}}$ for each aggregator are highlighted in bold (see below).

Aggregator (A)	t	0.01	0.05	0.10	0.20	0.30	0.40
TooManyFiringCat.	FPR	-	-	76.8	-	52.9	-
	FNR	-	-	0.0	-	0.0	-
TooManyFiringSens.	FPR	76.8	55.2	39.0	18.1	10.1	**5.4**
	FNR	0.0	0.0	0.0	0.0	4.5	**5.7**
TooManyFiringInp.	FPR	65.7	**11.4**	5.0	1.3	0.0	0.0
	FNR	0.0	**5.2**	13.3	27.8	74.1	94.1
Aggregator (A)	**t**	**0.50**	**0.60**	**0.70**	**0.90**	**0.95**	
TooManyFiringCat.	FPR	29.4	-	4.2	**0.0**	-	
	FNR	0.0	-	0.0	**1.9**	-	
TooManyFiringSens.	FPR	2.8	1.6	0.7	0.0	0.0	
	FNR	9.2	12.9	13.3	49.8	91.6	
TooManyFiringInp.	FPR	0.0	-	-	-	-	
	FNR	100.0	-	-	-	-	

Being $l = 50$ the size of the learning sequence, these rates mean that respectively $0, 1, 3, 5, 10, 18, 25, 38$ positive readings have been inserted in the learning sequence.

Table 5.2 shows FPR and FNR for the 3 aggregators with varying values of the corruption rate r. These results refer to the optimal threshold t_{opt} determined as explained above for $r = 0$.

Not surprisingly, increasing the corruption rate results in an increment of FNR for each aggregator. In other words, the more corrupted the learning sequence, the less sensitive to attacks the aggregator. Increasing the corruption rate also results in a decrease of FPR, due to the fact that the learning sequence, and hence the profile, becomes less and less page-specific. Although performance appears to quickly become unacceptable, varying the threshold may greatly help, as clarified in the following. The reason is because the above data corresponds to a threshold t that is optimal for an *uncorrupted* learning sequence, but this value is not necessarily optimal for a *corrupted* one.

A more general characterization of the performance of each aggregator is given in Figure 5.2, which plots A_{ROC} as a function of the corruption rate r. As such, each point in this graph provides a performance index capturing *all* possible values for the threshold t. We found that A_{ROC} does *not* decrease monotonically when the corruption rate increases. On the contrary there is an A_{ROC} increase when $r < 0.05$, the entity of the improvement being dependent on the specific aggregator. In other words, a very small corruption in the learning sequence is beneficial for all aggregators from the A_{ROC}

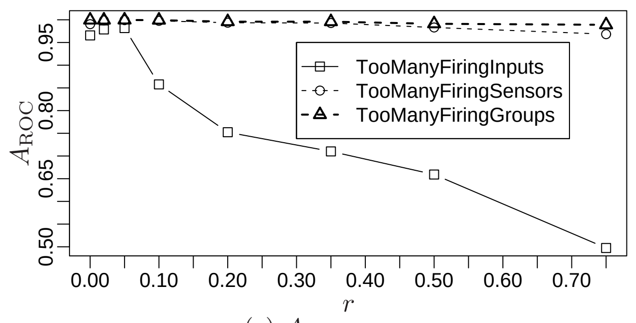
Table 5.2: FPR and FNR, obtained with $t = t_{\text{opt}}$ (see Table 5.1) and presented in percentage, for the aggregator with different values for r.

	TooMany-FiringCategories		TooMany-FiringSensors		TooMany-FiringInputs	
	FPR	FNR	FPR	FNR	FPR	FNR
r	$t = t_{\text{opt}} = 0.9$		$t = t_{\text{opt}} = 0.4$		$t = t_{\text{opt}} = 0.05$	
0.00	0.0	1.9	5.4	5.7	11.4	5.2
0.02	0.0	73.4	0.1	12.4	6.5	5.6
0.05	0.0	77.6	0.1	12.9	5.5	6.3
0.10	0.0	83.9	0.0	69.5	2.5	75.8
0.20	0.0	87.5	0.1	99.9	2.5	99.4
0.35	0.0	86.7	0.1	99.9	2.6	99.6
0.50	0.0	77.6	0.1	99.7	2.8	99.6
0.75	0.0	87.7	0.3	99.8	11.3	99.4

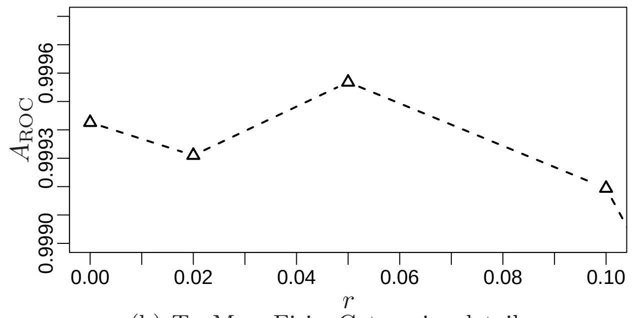
point of view. This suggests that under a modest corruption there is some point of the ROC curve of each aggregator—i.e., some value of t—for which FPR and FNR are acceptable and, maybe, even slightly better than with an uncorrupted learning sequence. Of course, finding that point would require the knowledge of the corruption rate. The slight increase in A_{ROC} is probably due to the fact that a small amount of noise (i.e., corruption) may balance the overfitting effect, which affects negatively FPR and may be an issue in pages that are less dynamic and whose corresponding learning sets are hence less representative.

The relation between FPR+FNR and t is shown in Figure 5.3, which plots one curve for each corruption rate. The lowest point of each curve corresponds to $t = t_{\text{opt}}$. We remark again that one could choose to minimize a different function of FPR and FNR. The essential issues of our arguments would not change, however. Figure 5.3 illustrates two important facts. First, the optimal threshold depends on the corruption rate. That is, a threshold optimized for an uncorrupted learning sequence is not necessarily the best threshold for a corrupted learning sequence. Second, performance with a corrupted learning sequence are not necessarily worse than with an uncorrupted learning sequence. For example, decreasing t improves the FPR+FNR index significantly, especially for the TooManyFiringSensors aggregator. Table 5.3 summarizes the improvements exhibited by the aggregators using the optimal value t_{opt} for two salient r values.

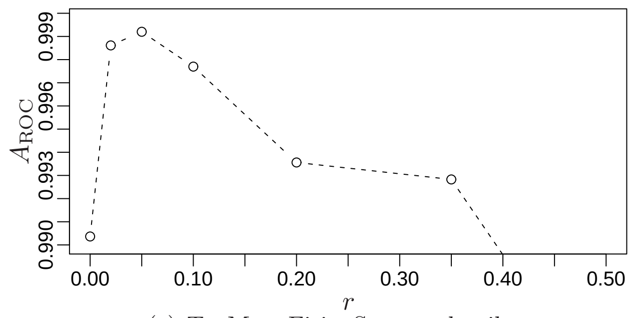
A key lesson from these experiments is that the aggregators may remain practically useful (i.e., they exhibit acceptable FPR and FNR) even in the presence of a moderate degree of corruption in the learning sequence. The problem is, turning this observation into a practically usable procedure is far from being immediate: one should know the corruption rate in order to select a suitable working point, but this is precisely the unknown entity.



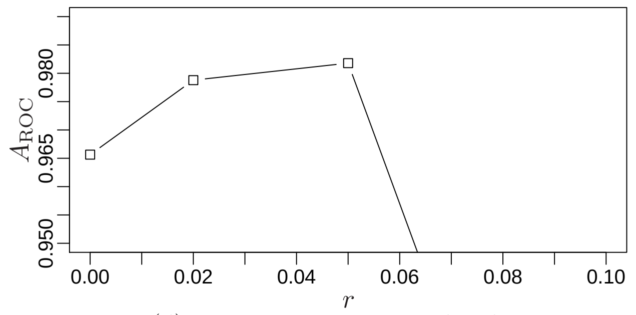
(a) A_{ROC} vs. r



(b) TooManyFiringCategories detail

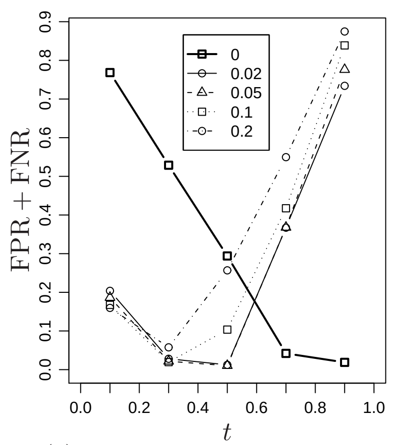


(c) TooManyFiringSensors detail

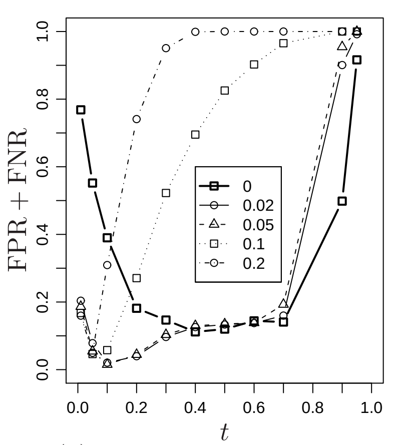


(d) TooManyFiringInputs detail

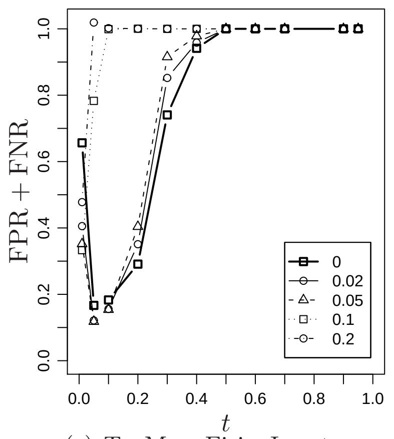
Figure 5.2: The area under the ROC curve (A_{ROC}) vs. the corruption rate r. Figures 5.2(b), 5.2(c) and 5.2(d) show salient A_{ROC} values for the 3 aggregators separately.



(a) TooManyFiringCategories



(b) TooManyFiringSensors



(c) TooManyFiringInputs

Figure 5.3: Effectiveness of aggregators, as sum of FPR and FNR, plotted vsthe aggregator normalized discrimination threshold t, for different modest corruption rates including $r = 0$. The lowest point of each curve corresponds to the optimal working point $t = t_{\text{opt}}$ in the given conditions.

Table 5.3: Aggregators optimal working point for two different corruption rates and corresponding FPR and FNR, presented in percentage.

Aggregator (A)	r	t_{opt}	FPR	FNR	FPR + FNR
TooManyFiringCategories	0.0	0.9	0.0	1.9	1.9
	0.05	0.5	0.2	0.8	1.0
TooManyFiringSensors	0.0	0.4	5.4	5.7	11.1
	0.05	0.1	1.5	0.0	1.5
TooManyFiringInputs	0.0	0.05	11.4	5.2	16.6
	0.05	0.05	5.5	6.3	11.8

5.4 A corruption detection procedure

We have seen in the previous section that the impact of the corruption rate r on FPR and FNR is not linear. In particular, it can be observed that the change in FPR and FNR is much sharper when r increases from 0 to 0.02 than when r increases from 0.02 to 0.05 (see Table 5.2). The effects on performance, thus, are much stronger when switching from a clean learning sequence to a corrupted learning sequence than with a moderate increase of a (non-zero) corruption rate. We performed a number of experiments, not shown here for space reasons, to verify that this phenomenon does occur in a broad range of operating conditions. We exploited the above observation for building a simple yet effective corruption detection procedure, which is presented below.

5.4.1 Description

The objective is to determine whether a given learning sequence S^0_{learning} is corrupted. The key idea is quite simple. We build three profiles, one with S^0_{learning} and the other with two learning sequences obtained by artificially corrupting S^0_{learning} with 1 or 3 positive readings. Then we measure performance of the three profiles on a *same* sequence S_{check}. If we observe a strong change in FPR and/or FNR when switching from the first profile to the other two profiles, then S^0_{learning} was probably clean, otherwise it was probably already corrupted.

In detail, let $S'_P = \{i^p_0, \dots, i^p_n\}$ be a set of n positive readings. We construct S_{check} with a mixture of genuine readings and positive readings. Then we proceed as follows:

1. we tune the aggregator on S^0_{learning}; we measure FPR^0 and FNR^0 on the check sequence S_{check};
2. for a given i^p_i in S'_P, we construct two learning sequence $S^{1,i}_{\text{learning}}$ and $S^{3,i}_{\text{learning}}$ by replacing respectively 1 and 3 random readings of S^0_{learning} with i^p_i; we tune the aggregator on these learning sequences; we measure the corresponding performance on S_{check} ($\text{FPR}^{1,i}$, $\text{FNR}^{1,i}$, $\text{FPR}^{3,i}$ and $\text{FNR}^{3,i}$);
3. we repeat the previous step for each i^p_i of S'_P and evaluate mean (FPR^η and FNR^η) and standard deviation (FPR^σ and FNR^σ) of the performance indexes.

The original learning sequence S^0_{learning} is deemed corrupted if and only if at least one of the following holds:

$$\text{FPR}^0 - \text{FPR}^\eta \geq m\text{FPR}^\sigma \tag{5.1}$$
$$\text{FNR}^0 - \text{FNR}^\eta \geq m\text{FNR}^\sigma \tag{5.2}$$

where m corresponds to a sensitivity parameter of the procedure.

5.4.2 Evaluation and results

We measured the effectiveness of our procedure as follows:

1. we generated an uncorrupted learning sequence;
2. we artificially corrupted this sequence with a positive reading repeated until the end of the sequence (much like Section 5.3.2);
3. then, we applied the procedure.

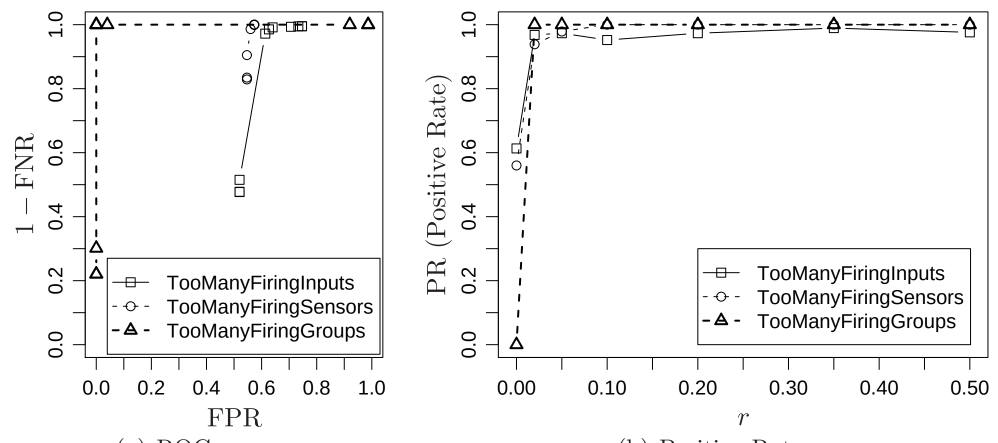
We experimented several corruption rates r: 0, 0.01, 0.05, 0.1, 0.2, 0.35, 0.5. For each learning sequence, S_{check} contained 50 positive readings and 50 negative readings of the page described by the learning sequence. For each pair $\langle r, \text{page} \rangle$, we repeated the test 25 times, with $N_S = 5$ different learning sequences at step 1 and $N_p = 5$ different positive readings at step 2.

Whenever the procedure stated that the learning sequence was corrupted, the test counted as a true positive if $r \neq 0$ and as a false positive otherwise. Whenever the procedure stated that the learning sequence was *not* corrupted, the test counted as a false negative if $r \neq 0$ and as a true negative otherwise.

Figure 5.4(a) shows the ROC curves, obtained experimenting with different values for m. It can be seen that with TooManyFiringSensors and TooManyFiringInputs the procedure exhibits unsatisfactory performance, in that FPR is never smaller than 0.6 irrespective of m (see also what follows). With TooManyFiringCategories, on the other hand, it appears to exhibit an ideal behavior.

Figure 5.4(b) plots the positive rate with $m = 1$ as a function of the corruption rate r (the optimum would correspond to a positive rate 0 for $r = 0$ and 1 otherwise). This figure clearly shows that the procedure achieves the optimum (at least in our benchmark) with the TooManyFiringCategories aggregator: it detects each corrupted learning sequence while not misclassifying any clean sequence. With the two other aggregators, on the other hand, it exhibits far too many false positives. We interpret this result as a consequence of the previous results in Table 5.2: when switching from a clean learning sequence to a corrupted one, the performance change is not sufficiently strong, with TooManyFiringSensors and TooManyFiringInputs.

Another important result from Figure 5.4(b) is that the detection accuracy of corrupted sequences is very high over the whole range of r.



(a) ROC curves (b) Positive Rate

Figure 5.4: Effectiveness of our corruption detection procedure applied to the 3 aggregators. Figure on the left shows ROC curves. Figure on the right plots positive rates for the procedure applied with $m = 1$.

Chapter 6

Genetic Programming for automatic defacement detection

In this chapter we describe our study about a novel web site defacement detection approach based on Genetic Programming (GP), an established evolutionary computation paradigm for automatic generation of algorithms. What makes GP particularly attractive in this context is that it does not rely on any domain-specific knowledge, whose description and synthesis is invariably a hard job.

6.1 Genetic Programming in a nutshell

Genetic Programming (GP) is an automatic method for creating computer programs by means of artificial evolution [35]. A *population* of computer programs are generated at random starting from a user-specified set of building blocks. Each such program constitutes a candidate solution for the problem and is called an *individual*. The user is also required to define a *fitness function* for the problem to be solved. This function quantifies the performance of any given program—i.e., any candidate solution—for that problem (more details will follow). Programs of the initial population that exhibit highest fitness are selected for producing a new population of programs. The new population is obtained by recombining the selected programs through certain genetic operators, such as "crossover" and "mutation". This process is iterated for some number of generations, until either a solution with perfect fitness is found or some termination criterion is satisfied, e.g., a predefined maximum number of generations have evolved. The evolutionary cycle is illustrated in Figure 6.1.

In many cases of practical interest individuals—i.e., programs—are simply formulas. Programs are usually represented as abstract syntax trees, where a branch node is an element from a *functions set* which may contain arithmetic, logic operators, elementary functions with at least one argument. A leaf node of the tree is instead an element from a *terminals set*, which usually contains variables, constants and functions with no

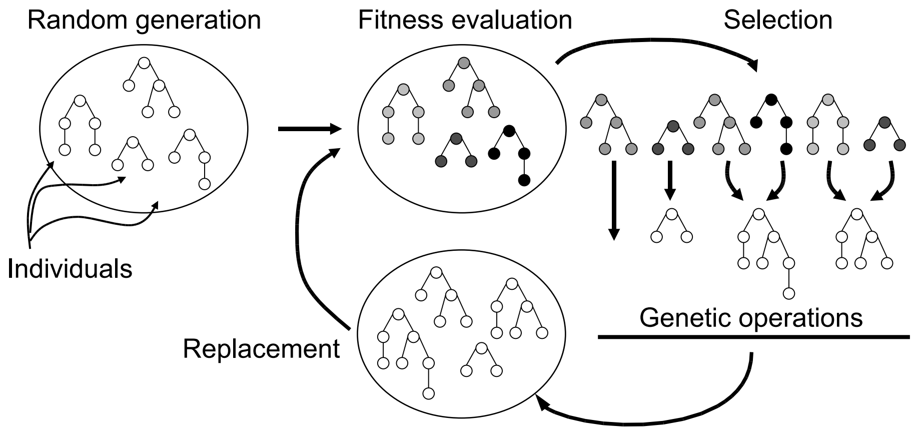


Figure 6.1: The evolutionary cycle.

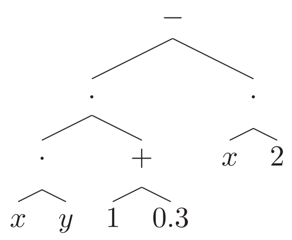


Figure 6.2: A tree representation of $1.3xy - 2x$.

arguments. Figure 6.2 shows a sample tree representing a simple mathematical formula. Functions set and terminals set constitute the previously mentioned building blocks to be specified by the user. Clearly, these blocks should be expressive enough to represent satisfactory solutions for the problem domain. The population size is also specified by the user and depends mainly on the perceived "difficulty" of the problem.

6.2 Related work

Several prior works have addressed the use of GP [80, 52], as well as other evolutionary computation techniques [75, 12], for network based or host based intrusion detection systems. What makes such approaches attractive is their ability to find automatically models capable of coping effectively with the huge amount of information to be handled [71]. We are not aware of any attempt of using such techniques for automatic detection of web defacements. One of the key differences between our scenario and such prior studies concerns the nature of the dataset used for training: in our case, it includes relatively few readings (some tens), each one composed by many attributes (or values), whereas in the network and host-based IDS fields, it usually includes much more readings (thousands and more), each one composed by few attributes.

6.3 Scenario

We use GP within the Goldrake prototype that we described in Chapter 3. In particular, we implemented an aggregator which works internally according to the GP process. The GP process is actually applied at the end of the learning phase. Each individual implements a function $F(v)$ of the output v of the refiner, for example (v^i denotes the i-th component of v):

$$F(v) = 12 - \max(v^{57}, v^{233}) + 2.7v^{1104} - \frac{v^{766}}{v^{1378}} \tag{6.1}$$

The building blocks used for constructing individuals are described below. The output of the aggregator for a given reading i_k is defined as follows (v_k denotes the refiner output for the reading i_k):

$$y_k = \begin{cases} \text{negative} & \text{if } F(v_k) < 0 \\ \text{positive} & \text{if } F(v_k) \geq 0 \end{cases} \tag{6.2}$$

Individuals, i.e., formulas, of the population are selected basing on their ability to solve the detection problem, which is measured by the fitness function. In this case, we want to maximize the aggregator ability to detect attacks and, at the same time, minimize the number of wrong detections. For this purpose, we define a fitness function in terms of *false positive rate* (FPR) and *false negative rate* (FNR), as follows:

1. we build a sequence S_{learning} of readings composed by readings of the observed page (S_l) and a sample set of attacks readings;
2. we count the number of false positives—i.e., genuine readings considered as attacks—and false negatives—i.e., attacks not detected—raised by F over S_{learning}, thus computing the respective FPR and FNR;
3. we finally set the fitness value $f(F)$ of the individual F as follows:

$$f(F) = \text{FPR}(S_{\text{learning}}) + \text{FNR}(S_{\text{learning}}) \tag{6.3}$$

This fitness definition is applied by the GP process to select best individuals and repeat the evolution cycle, until either of the following holds: (1) a formula F for which $f(F) = 0$ is found or (2) more than $n_{g,\max} = 100$ generations have evolved. In the latter case, the individual with the best (lowest) fitness value is selected.

The building blocks for individuals are:

- a terminals set composed of $\mathcal{C} = \{0, 0.1, 1\}$, a specified set of constants, and $\mathcal{V}$, a specified set of independent variables from the output of the refiner;
- a functions set composed of $\mathcal{F}$, a specified set of functions.

We experimented with different sets $\mathcal{V}$ and $\mathcal{F}$ in order to gain insights into the actual applicability of GP to this task.

Concerning $\mathcal{F}$ we experimented with various subsets of (see Section 6.4.2):

$$\mathcal{F}_{\text{all}} = \{+, -, \cdot, \div, \smallsetminus, \min, \max, \leq, \geq\} \tag{6.4}$$

where $\smallsetminus$ represents the unary minus, and $\leq$ and $\geq$ returns 1 or 0 depending on whether the relation is true or not. All functions take two arguments, except for the unary minus.

Concerning $\mathcal{V}$ we experimented with various subsets of the set composed of all elements of v, the vector output by the refiner. To this end, we applied a *feature selection* algorithm for deciding which elements of v should be included in $\mathcal{V}$. The algorithm is described in the next section. Note that elements in v not included in $\mathcal{V}$ will have no influence whatsoever on the decision taken by the aggregator, because no individual of the GP process will ever include such elements.

6.3.1 Feature selection

The feature selection algorithm is aimed at selecting those v elements which seem to indeed have significance in the decision and works as follows.

Let $S_{\text{learning}} = \{v_1, \dots, v_n\}$ be the learning sequence, including all genuine readings and all simulated attacks. Let X_i denote the random variable whose values are the values of the i-th element of v (i.e., v^i) across all readings of S_{learning}. Recall that there are 1466 such variables because this is the size of v. Let Y be the random variable describing the desired values for the aggregator: $Y = 0$, $\forall$ genuine reading $\in S_{\text{learning}}$; $Y = 1$ otherwise, i.e., $\forall$ simulated attack reading $\in S_{\text{learning}}$. We computed the absolute correlation c_i of each X_i with Y and, for each pair $\langle X_i, X_j \rangle$, the absolute correlation $c_{i,j}$ between X_i and X_j. Finally, we executed the following iterative procedure, starting from a set of unselected indexes $I_U = \{1, \dots, 1466\}$ and an empty set of selected indexes $I_S = \emptyset$:

1. we selected the element $i \in I_U$ with the greatest c_i and moved it from I_U to I_S;

2. $\forall j \in I_U$, we set $c_j := c_j - c_{i,j}$.

We repeated the two steps until a predefined size s for I_S is reached. Set $\mathcal{V}$ will include only those elements of vector v whose indexes are in I_S. In other words, we take into account only those indexes with maximal correlation with the desired output (step 1), attempting to filter out any redundant information (step 2).

6.4 Experimental evaluation

6.4.1 Dataset

In order to perform our experiments, we used a reduced version of the dataset collected for testing Goldrake (see Section 4.2). It is composed by readings collected observing 15 web pages for about one month, downloading a reading for each page every 6 hours, until totaling 125 readings for each web page. These readings compose the *negatives sequences*—one negative sequence S_N for each page: we visually inspected them in order to confirm the assumption that they are all genuine. We also used a single *positives sequence* S_P composed by 75 readings extracted from a publicly available defacements archive (http://www.zone-h.org).

6.4.2 Methodology

In order to set a baseline for assessing the performance of GP, we injected the very same dataset to the TooManyFiringCategories aggregator, that we described earlier in Section 3.3.2. Recall that this aggregator implements a form of anomaly detection based on domain-specific knowledge. Moreover, it makes use of a learning sequence that does not include any attack, thus it does not exploit any information related to positive readings. The GP-based aggregator, in contrast, does use such information. Note also that our existing aggregator takes into account all the 1466 elements output by the refiner, whereas GP-based aggregator considers only those elements chosen by the feature selection algorithm.

We generated 25 different GP-based aggregators, by varying the number s of selected features in 10, 20, 50, 100, 1466 (thus including the case in which we did not discard any feature) and the functions set $\mathcal{F}$, choosing among these 5 subsets:

- $\mathcal{F}_1 = \{+, -\}$
- $\mathcal{F}_2 = \{+, -, \cdot, \div, \diagdown\}$
- $\mathcal{F}_4 = \{+, -, \cdot, \div, \diagdown, \leq, \geq\}$
- $\mathcal{F}_3 = \{+, -, \cdot, \div, \diagdown, \min, \max\}$
- $\mathcal{F}_5 = \{+, -, \cdot, \div, \diagdown, \min, \max, \leq, \geq\}$

We used FPR and FNR as performance indexes, that we evaluated as follows. First, we built a sequence S'_P of positive readings composed by the first 20 readings of S_P. Then, for each page p, we built the learning sequence S_{learning} and a testing sequence S_{testing} as follows:

1. We split the corresponding S_N in two portions S_l and S_t, composed by 50 and 75 readings respectively.
2. We built the learning sequence S_{learning} appending S'_P to S_l.
3. We built the testing sequence S_{testing} appending S_P to S_t.

Finally, we tuned the aggregator being tested on S_{learning} and we tested it on S_{testing}. To this end, we counted the number of false negatives—i.e., missed detections—and the number of false positives—i.e., legitimate readings being flagged as attacks. As already pointed out, the anomaly-based aggregator executes the learning phase using only on S_l and ignoring S'_P.

In the next sections we present FPR and FNR for each aggregator, averaged across the 15 pages of our dataset. GP-based aggregators will be denoted as GP-s-$\mathcal{F}_i$, where s is the number of selected features and $\mathcal{F}_i$ is the specific set of functions being used; anomaly-based aggregator will be denoted as Anomaly.

6.4.3 Results

Table 6.1 summarizes our results. The table shows FPR, FNR and the fitness exhibited by the individual selected to implement the GP-based aggregator (the meaning of the three other columns is discussed below). The aggregator with best performance, in terms of FPR + FNR, is highlighted. It can be seen that the GP process is quite robust with respect to variations in s and $\mathcal{F}$. Almost all GP-based aggregators exhibit a FPR lower than 0.36% and a FNR lower than 1.87%. The anomaly-based aggregator—i.e., the comparison baseline—exhibits a slightly higher FPR (1.42%) and a slightly lower FNR (0.09%). In general, the genetic programming approach seems to be quite effective for detecting web defacements.

We analyzed GP-based aggregators also by looking at the number of generations n_g that have evolved for finding the best individual and the complexity of the corresponding abstract syntax tree, in terms of number of nodes t_s and height t_h (these data are also shown in Table 6.1). We found that formulas tended to be quite simple, i.e., the corresponding trees exhibited low t_s and t_h. For example, in many cases the produced formula looked like the following:

$$y = 1 - (v^{188} - v^{1223}) \tag{6.5}$$

where v^i is the i-th element of the vector output by the refiner and for which $t_s = 3$ and $t_h = 2$. We also found, to our surprise, that $n_g = 1$ in most cases. This means that generating some random formulas (500 in our experiments) from the available functions and terminals sets suffices to find a formula with perfect fitness—i.e., one that exhibits 0 false positives and 0 false negatives over the learning sequence. We believe this depends on the high correlation between some elements of the vector output by the refiner (i.e., some v^i) and the desired output. Since the feature selection chooses elements based on their correlation with the desired output, most of the variables available to GP will likely have an high correlation with output.

Our domain knowledge, however, suggests that the simple formulas found by the GP process could not be very effective in a real scenario. Since they take into account very few variables, an attack focusing on the many variables ignored by the corresponding GP aggregators would go undetected. This consideration convinced us to develop a more demanding test-bed, as follows.

6.4.4 Results with "shuffled" dataset

In this additional set of experiments, we augmented the set of positive readings for any given page p_i by including genuine readings of *other* pages. While the previous experiments evaluated the ability to detect manifest attacks (defacements extracted

Table 6.1: Performance indexes. FPR, FNR and f are expressed in percentage.

Aggregator	FPR	FNR	f	n_g	t_s	t_h
Anomaly	1.42	0.09	-	-	-	-
GP-10-$\mathcal{F}_1$	0.00	0.71	0.0	1.0	17.0	2.7
GP-10-$\mathcal{F}_2$	0.09	0.98	0.0	1.0	23.3	3.7
GP-10-$\mathcal{F}_3$	0.09	0.62	0.0	1.1	20.4	3.5
GP-10-$\mathcal{F}_4$	4.53	0.44	0.0	1.0	20.7	3.7
GP-10-$\mathcal{F}_5$	0.09	0.89	0.0	1.0	27.9	3.9
GP-20-$\mathcal{F}_1$	0.09	1.16	0.0	1.0	18.2	2.6
GP-20-$\mathcal{F}_2$	0.18	1.33	0.0	1.0	12.8	2.4
GP-20-$\mathcal{F}_3$	0.36	0.80	0.0	1.0	20.1	3.1
GP-20-$\mathcal{F}_4$	0.09	0.89	0.0	1.0	36.5	4.2
GP-20-$\mathcal{F}_5$	0.00	0.89	0.0	1.0	39.5	3.9
GP-50-$\mathcal{F}_1$	0.00	1.24	0.0	1.0	5.1	1.6
GP-50-$\mathcal{F}_2$	0.09	0.98	0.0	1.0	20.4	2.9
GP-50-$\mathcal{F}_3$	0.36	0.98	0.0	1.0	19.3	2.9
GP-50-$\mathcal{F}_4$	0.18	0.89	0.0	1.0	15.4	3.1
GP-50-$\mathcal{F}_5$	0.18	0.27	0.0	1.0	29.4	3.0
GP-100-$\mathcal{F}_1$	0.09	1.16	0.0	1.0	15.5	2.1
GP-100-$\mathcal{F}_2$	0.09	1.33	0.0	1.1	11.4	2.2
GP-100-$\mathcal{F}_3$	0.00	1.87	0.0	1.3	14.1	3.1
GP-100-$\mathcal{F}_4$	**0.09**	**0.27**	**0.0**	**1.1**	**18.7**	**3.1**
GP-100-$\mathcal{F}_5$	0.09	1.33	0.0	1.2	15.4	2.6
GP-1466-$\mathcal{F}_1$	0.00	0.80	0.0	1.0	8.9	1.9
GP-1466-$\mathcal{F}_2$	0.18	0.44	0.0	1.0	6.1	2.3
GP-1466-$\mathcal{F}_3$	0.18	0.98	0.0	1.2	5.3	1.5
GP-1466-$\mathcal{F}_4$	0.18	1.87	0.0	1.2	11.2	1.8
GP-1466-$\mathcal{F}_5$	3.73	0.18	0.0	1.4	9.9	2.1

from Zone-H), here we also test the ability to detect innocent-looking pages that are different from the usual appearance of p_i. More in detail, for a given page p_i we defined a sequence S^o_{learning} composed by 14 genuine readings of the *other* pages (one for each other page $p_j \neq p_i$) and a sequence S^o_{testing} composed by 70 readings of the other pages (5 readings for each other page $p_j \neq p_i$, such that S^o_{learning} and S^o_{testing} have no common readings). Then, we included S^o_{learning} in S_{learning} and S^o_{testing} in S_{testing} (we omit the obvious details for brevity). Clearly, readings in S^o_{learning} were labeled as positives and readings in S^o_{testing} should raise an alarm.

Table 6.2 presents the results for this test-bed. The anomaly-based aggregator now exhibits a slightly higher FNR; FPR remains unchanged, which is not surprising because this aggregator uses only the negative readings of the learning sequence and these, like the negative portion of the testing sequence, are the same as before. We note that the anomaly-based aggregator is very effective also in this new setting, in that it still exhibits a very low FPR while being capable of flagging as positive most of the genuine readings of *other* pages.

Concerning GP-based aggregators, Table 6.2 suggests several important considerations. In general the approach seems now to be influenced by the number s of variables selected: taking more variables into account lead to better performance, in terms of $\text{FPR} + \text{FNR}$. Interestingly, the best result is obtained with $s = 1466$, i.e., with *all* variables available to the GP process. Moreover, there are situations in which the fitness f of the selected formula is no longer perfect (i.e., equal to 0). This means that, in these situations, the GP process is no longer able to to find a formula that exhibits $f = \text{FPR} + \text{FNR} = 0$ on the learning sequence. Note that this phenomenon tends to occur with low values of s. Finally, values of t_s, t_h and, especially, n_g, are greater than in the previous test-bed, which confirms that GP process is indeed engaged. Several generations are necessary to find a satisfactory formula and the formula is more complex than those previously found, in terms of size and height of the corresponding abstract tree.

Figure 6.3 shows results of Table 6.2 in a graphical way and visually confirms that the GP approach is quite robust in respect to the specific set of functions being used but is sensible to the number s of selected features.

Finally, we compared the computation times for learning and monitoring phases obtained with GP-based approach against those of anomaly-based approach. The former takes about 100secs for performing the tuning procedure (of which about 5secs are used for the features selection) and about 500μsecs for evaluating one single reading in the monitoring phase; the latter takes about 10msecs for the tuning procedure and about 100μsecs for evaluating one single reading in the monitoring phase. These numbers are obtained with a dual AMD Opteron 64 with 8GB RAM running a Sun JVM 1.5 on a Linux OS.

Table 6.2: Performance indexes with the new test-bed. FPR, FNR and f are expressed in percentage.

Aggregator	FPR	FNR	f	n_g	t_s	t_h
Anomaly	1.42	2.39	-	-	-	-
GP-10-$\mathcal{F}_1$	9.87	2.48	0.4	35.5	83.8	7.2
GP-10-$\mathcal{F}_2$	9.24	1.84	0.0	14.3	69.1	7.5
GP-10-$\mathcal{F}_3$	9.24	1.29	0.1	35.7	66.4	8.0
GP-10-$\mathcal{F}_4$	11.38	1.98	0.1	24.1	93.1	7.9
GP-10-$\mathcal{F}_5$	7.73	1.52	0.1	30.5	66.1	6.9
GP-20-$\mathcal{F}_1$	23.38	2.30	0.1	16.9	54.2	5.3
GP-20-$\mathcal{F}_2$	16.44	1.61	0.0	10.8	68.3	6.2
GP-20-$\mathcal{F}_3$	13.51	1.33	0.0	16.4	52.1	6.2
GP-20-$\mathcal{F}_4$	17.87	1.38	0.0	14.6	56.3	6.3
GP-20-$\mathcal{F}_5$	17.87	0.55	0.0	19.5	70.4	6.5
GP-50-$\mathcal{F}_1$	14.76	1.56	0.1	23.7	43.1	4.4
GP-50-$\mathcal{F}_2$	13.16	1.61	0.0	4.7	45.0	5.4
GP-50-$\mathcal{F}_3$	18.58	0.83	0.0	11.7	32.4	5.4
GP-50-$\mathcal{F}_4$	7.56	1.52	0.0	12.5	41.1	6.0
GP-50-$\mathcal{F}_5$	11.47	1.66	0.0	16.1	71.2	6.8
GP-100-$\mathcal{F}_1$	0.62	2.30	0.0	29.4	51.5	4.5
GP-100-$\mathcal{F}_2$	5.51	1.38	0.0	10.5	25.6	4.1
GP-100-$\mathcal{F}_3$	6.93	0.55	0.0	16.4	33.9	4.8
GP-100-$\mathcal{F}_4$	5.87	1.61	0.0	10.2	40.3	5.1
GP-100-$\mathcal{F}_5$	12.09	1.52	0.0	17.7	31.9	5.2
GP-1466-$\mathcal{F}_1$	0.18	1.38	0.0	21.0	37.4	3.8
GP-1466-$\mathcal{F}_2$	0.44	1.10	0.0	18.5	24.8	4.1
GP-1466-$\mathcal{F}_3$	**0.71**	**0.64**	**0.0**	**15.1**	**30.1**	**4.7**
GP-1466-$\mathcal{F}_4$	5.69	1.06	0.0	19.7	27.9	4.7
GP-1466-$\mathcal{F}_5$	0.98	1.24	0.0	16.5	69.9	5.5

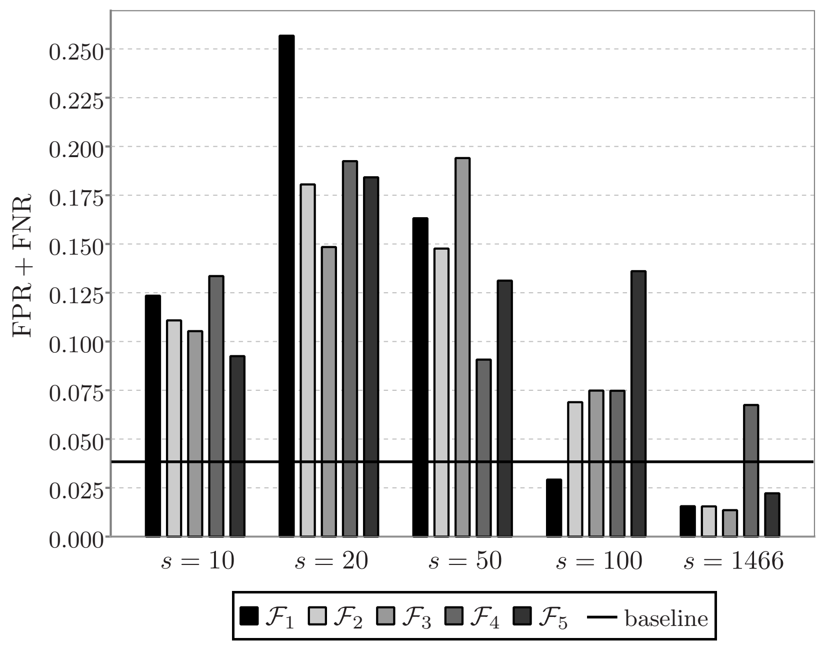


Figure 6.3: Sum of FPR and FNR for different parameter combinations.

Chapter 7

A comparison of other anomaly detection techniques

In the previous chapters we presented our approach and its implementation for detecting automatically web site defacements on a large scale. Our strategy incorporates domain-specific knowledge about the nature of web contents and, in particular, does not build upon the abundant literature about anomaly-based intrusion detection systems, e.g., [16, 23, 36, 81, 54, 37]. On the one hand, we do not treat all observed features as being equivalent, but rather we distinguish between internal links and external links, we analyze the average size of pages and words, we reason about HTML tags recurring in each reading of a page, we consider that when the number of words in a page increases then the number of characters should increases as well, and so on. On the other hand, we tuned the numerous page-specific parameters in a very simple way—either to constant values or to predefined functions of mean and standard deviation of certain measures. We did not make any assumptions about statistical properties and probability distributions of the observed features.

In this chapter we broaden our analysis and compare our approach with anomaly detection techniques that have been proposed earlier for host/network-based intrusion detection systems [61, 6, 45, 28, 57, 5]. This study enables gaining further insights into the problem of automatic detection of web defacements. We want to ascertain whether techniques for anomaly-based intrusion detection may be applied also to anomaly detection of web defacements and we want to assess pros and cons of incorporating domain knowledge into the detection algorithm.

Interestingly, the anomaly detection techniques that we considered are often used inside IDSs to analyze the stream of data collected observing inputs and outputs of the monitored system. In the case of defacement detection, instead, we work on externally observable state, i.e., we observe a web page. Working on the inputs has the potential to detect any malicious activity promptly and even prevent it completely. In practice, this potential is quite hard to exploit due to the large number of false alarms generated if the rate of data analyzed is very high. Working on the external state, on the other hand, may only detect intrusions after they have occurred.

In the comparative evaluation of techniques presented in the next sections, we set as a baseline the TooManyFiringCategories aggregator which is presented in Section 3.3.2.

7.1 Anomaly detection techniques

In this section we briefly describe the techniques that we used in this comparative experimental evaluation: all these techniques have been proposed and evaluated for detecting intrusions in host or network based IDSs. Each technique is an algorithm aimed at producing a binary classification of an item expressed as a numeric vector (or point) $p \in \mathbb{R}^n$.

We incorporated each one of them in our framework by implementing a suitable aggregator, that is, they are all based on the same refiner and apply different criteria for classifying a reading. Each technique constructs a profile with a technique-specific method, basing on the readings contained in a learning sequence S. The learning sequence is composed by both negative readings (S^-) and positive readings (S^+). However, only one of the methods analyzed (Support Vector Machines) indeed make use of S^+. For all the other methods $S = S^-$ and $S^+ = \emptyset$.

During our preliminary experiments, we noticed that for all the techniques, except for TooManyFiringCategories, performance were poor in terms of both detection effectiveness and computational effort. We quickly realized that for these techniques the size of the vector v output by the refiner was excessive from both points of view. Hence, for all the techniques except for TooManyFiringCategories, we applied the *feature selection* algorithm described in Section 6.3.1 to reduce v size. Note that we selected for each technique the maximum value of s that appeared to deliver acceptable performance: we set $s = 10$ for K-th Nearest, LOF and Hotelling and $s = 20$ for the others. The feature selection algorithm is applied once for each web page.

7.1.1 K-th Nearest

This technique [61, 41, 32] is distance-based, often computed using Euclidean metric. Let k be an integer positive number and p the investigated point; we define the k-th nearest distance $D^k(p)$ as the distance $d(p, o)$ where o is a generic point of S^- such that:

1. for at least k points $o' \in S^-$ it holds that $d(p, o') \leq d(p, o)$, and

2. for at most $k - 1$ points $o' \in S^-$ it holds that $d(p, o') < d(p, o)$

We define a point p as a positive if $D^k(p)$ is greater than a provided threshold t.

In our experiments we used the Euclidean distance for d and we set $k = 3$ and $t = 1.01$.

7.1.2 Local Outlier Factor

Local Outlier Factor (from here on LOF) [6, 41] is an extension to the k-th nearest distance, assigning to each evaluated point an outlying degree. Let p be the investigated point; then the LOF value is computed as follows:

1. compute $D^k(p)$ and define *k-distance neighborhood* $N_k(p)$ as a set containing all the points $o \in S^-$ such that $d(p, o) \leq D^k(p)$;

2. define reachability distance $\text{reach-dist}(o, p) = \max\{D^k(p), d(o, p)\}$;

3. compute local reachability density $\text{lrd}(p)$ as the inverse of the average reachability distance of m points belonging to $N_k(p)$;

4. finally, LOF value $\text{LOF}(p)$ is defined as the average of the ratios of $\text{lrd}(o)$, with $o \in N_k(p)$, and $\text{lrd}(p)$;

A point p is defined as a positive if $\text{LOF}(p) \notin \left[\frac{1}{1+\epsilon}, 1+\epsilon\right]$, where ϵ represents a threshold. In our experiments we used the Euclidean distance for d.

7.1.3 Mahalanobis Distance

This is a distance measure based on the correlation between variables, providing a metric where all the points having the same distance lie on an ellipsoid instead of a sphere [45, 41].

Let C be the covariance matrix composed by elements of S^- and p the investigated point; then Mahalanobis distance is defined as:

$$D_M(p) = \sqrt{(p-\mu)^T C^{-1} (p-\mu)} \tag{7.1}$$

where μ is the vector of the averages of S^- vectors, i.e., $\mu = \{\mu^1, \ldots, \mu^n\}$ with $\mu^i = \frac{1}{N} \sum_k v_k^i$ and being N the size of S^-. If C is a singular matrix, we slightly modify it until it becomes non-singular by adding a small value to its diagonal.

We define a point as positive if $D_M(p) > \max\{D_M(o) | o \in S^-\} + t$, where t is a predefined threshold. In our experiments we set $t = 1.5$.

7.1.4 Hotelling's T-Square

Hotelling's T-Square method [28, 76, 78] is very similar to Mahalanobis distance.

Let C be the covariance matrix, computed as before; Hotelling's T-Square statistic is defined as:

$$t^2(p) = m(p-\mu)^T C^{-1} (p-\mu) \tag{7.2}$$

where m is the size of S^- and μ is the same as above.

We define a point p as positive if $t^2(p) > \max\{t^2(o) | o \in D^-\} + t$, where t is a predefined threshold. In our experiments we set $t = 5$.

7.1.5 Parzen Windows

Parzen Windows [57, 79, 32] provide a method to estimate the probability density function of a random variable.

Let $p = \{x_1, x_2, \ldots, x_n\} \in \mathbb{R}^n$ be the investigated point and X_i a random variable representing the i-th component of p, i.e., x_i.

Let $w(x)$ (also named *window function*) be a density function such that its volume is $V_0 = \int_{-\infty}^{+\infty} f(x)\, dx$; we considered two functions:

$$\text{Gaussian} : w(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

$$\text{Pulse} : w(x) = \begin{cases} 1 & \text{if } -\alpha \leq x \leq \alpha \\ 0 & \text{otherwise} \end{cases}$$

Then, we approximate the true density function $f(x_i)$ of the random variable X_i representing the i-th component of the point by (we denote with x_i^k the value of the i-th component of the k-th point of S^-):

$$\tilde{f}(x_i) = \frac{1}{n} \sum_{k=1}^{n} \frac{1}{V_k} w\left(\frac{x_i - x_i^k}{V_k}\right) \tag{7.3}$$

where $V_k = \frac{V_0}{\ln k}$; note that the weight term V_k decrease for older values (i.e., for points of S^- with an higher k).

We say that a component x_i of p is anomalous if and only if $\tilde{f}(x_i) < t_1$. We define a point p as positive if at least a percentage of components greater than t_2 is anomalous. In other words, with this method a probability distribution is estimated for each component of the input vector using its values along the learning sequence; then an alarm is raised if too many components seem not to be in accord with their estimated distribution.

In our experiments, we set $\sigma = 1$, $t_1 = 0.1$ and $t_2 = 7.5\%$ for Parzen Gaussian and $\alpha = 0.25$, $t_1 = 0.3$ and $t_2 = 10\%$ for Parzen Pulse.

7.1.6 Support Vector Machines

Support Vector Machines (SVM) [5, 51, 41] use hyperplanes to maximally separate N classes of data. In anomaly detection, we use only $N = 2$ classes of objects, positives and negatives. This technique uses a kernel function to compute the hyperplanes using both readings of S^- and S^+. In our experiments we used the *Radial Basis Function* (as part of the `libsvm` implementation [9]).

Once the hyperplanes are defined, a point p is considered as a positive if it is contained in the corresponding region.

7.2 Experimental evaluation

7.2.1 Dataset

In order to perform our experiments, we used a reduced version of the dataset collected for testing Goldrake (see Section 4.2). It has composed observing 15 web pages for two months, collecting a reading for each page every 6 hours, thus totaling a *negative sequence* of 240 readings for each web page. We visually inspected them in order to confirm the assumption that they are all genuine.

We also collected an *attack archive* composed by 95 readings extracted from a publicly available defacements archive (http://www.zone-h.org).

7.2.2 Methodology

We used False Positive Ratio (FPR) and False Negative Ratio (FNR) as performance indexes computing average, maximum and minimum values among web pages.

We proceeded as follows. For each page:

1. we built a sequence S^+ of positive readings composed by the first 20 elements of the attack archive;
2. we built a sequence S^- of negative readings composed by the first 50 elements of the corresponding negative sequence;
3. we built the learning sequence S by joining S^+ and S^-;
4. we trained each aggregator on S (recall that only one aggregator actually looks at S^+, as pointed out in Section 7.1).

Then, for each page:

1. we built a testing sequence S_t by joining a sequence S_t^-, composed by the remaining 190 readings of the corresponding negative sequence, and a sequence S_t^+, composed by the remaining 75 readings of the attack archive;
2. we fed the aggregator with each reading of S_t, as if it was the first reading to be evaluated after the learning phase, counting false positives and false negatives.

The previous experiments were done by constructing the profile of each web page only once, that is, by locking the internal state of each aggregator. We also experimented without locking the internal state, as follows. After the initial learning phase, whenever a reading of S_t^- was evaluated, we added that reading to S^- and removed the oldest reading from S^- (in other words, we used a sliding window of the 50 most recent readings of S^-); then, a new profile was immediately computed using the updated S^- (S^+ is never changed). In this way, we enabled a sort of continuous *retuning* that allowed each aggregator to keep the corresponding profile in sync with the web page, even in the long term. For the sole TooManyFiringCategories aggregator, the retuning was performed every 10 readings, instead of every reading. We computed FNR by injecting all the 75 readings of S_t^+ at the end of the test.

Table 7.1: Short term effectiveness with locked internal state.

	FPR (%)			FNR (%)		
Aggregator	avg	max	min	avg	max	min
TooManyFiringCategories	1.3	13.3	0.0	0.1	1.3	0.0
K-th Nearest	0.0	0.0	0.0	0.1	1.3	0.0
Local Outlier Factor	6.6	94.7	0.0	0.3	4.0	0.0
Hotelling	10.9	94.7	0.0	0.0	0.0	0.0
Mahalanobis	11.5	94.7	0.0	0.2	1.3	0.0
Parzen Pulse	1.2	16.0	0.0	0.0	0.0	0.0
Parzen Gaussian	4.1	40.0	0.0	0.0	0.0	0.0
Support Vector Machines	0.0	0.0	0.0	0.0	0.0	0.0

7.2.3 Results with internal state locked

In this section, we present FPR (with average, min and max values among the 15 web pages) that we obtained experimenting with the internal state locked, evaluated on the first 75 readings of S_t^- and on all the 190 readings of S_t^-. In other words, we assessed the effectiveness of each technique separately on the short term and on the long term—about 19 and 48 days respectively. We also present FNR (again with average, min and max values among the 15 web pages) computed on all readings of S_t^+.

Table 7.1 shows results obtained in the short term (i.e., after 75 negative readings): FNR values suggest that all the algorithms proved to be effective when detecting defacements. On the other hand, they behaved differently when analyzing genuine readings.

TooManyFiringCategories performed well on many web pages, although on some of them it exhibited very high FPR; Mahalanobis, Hotelling and LOF did not score well, being unable to classify genuine pages for many pages.

Both Parzen methods proved to be acceptably effective on many pages, although on some of them they worked as badly as Mahalanobis and Hotelling.

An excellent result comes from K-th Nearest and Support Vector Machines: both techniques managed to correctly classify all the negative readings, while still detecting a large amount of attacks.

Table 7.2 shows results obtained in the long term (i.e., after 190 negative readings), again with locked internal state. FNR is the same as Table 7.1, since both aggregator internal state and the positive testing sequence S_t^+ remain the same. Results in terms of FPR are slightly worse for all the evaluated techniques, as expected; the only aggregator that managed to perform almost as good as in short term is the one based on the Support Vector Machines. As a matter of fact, both K-th Nearest and Pulse Parzen managed to maintain a low FPR, but raised many false alarms on some web pages.

Figure 7.1 shows FPR vs. time, expressed as the index n of the reading of S_t^- (Table 7.1 corresponds to values for $n = 75$, whereas Table 7.2 corresponds to values for $n = 190$). The figure makes it easy to assess the effectiveness of each technique as

Table 7.2: Long term effectiveness with locked internal state.

	FPR (%)			FNR (%)		
Aggregator	avg	max	min	avg	max	min
TooManyFiringCategories	2.7	26.3	0.0	0.1	1.3	0.0
K-th Nearest	1.8	18.9	0.0	0.1	1.3	0.0
Local Outlier Factor	11.0	97.9	0.0	0.3	4.0	0.0
Hotelling	14.7	97.9	0.0	0.0	0.0	0.0
Mahalanobis	16.2	97.9	0.0	0.2	1.3	0.0
Parzen Pulse	1.3	15.3	0.0	0.0	0.0	0.0
Parzen Gaussian	4.5	40.0	0.0	0.0	0.0	0.0
Support Vector Machines	0.0	0.5	0.0	0.0	0.0	0.0

time goes by: many aggregators perform well at the beginning but then start raising a substantial amount of false alarms. As it turns out from these results, thus, building the profile of a web page based on a single observation period is not enough because after some time such a profile is no longer representative of the current content and appearance of the web page. Hotelling, Mahalanobis and LOF curves show that these techniques hardly cope with the fact that the profile of web page is getting older. Other techniques seem to be more robust in this sense.

7.2.4 Results with retuning

In this section we present results of experiments done without locking the internal state of the aggregator, i.e., while retuning each aggregator at each reading. As described in Section 7.2.2, we used as S^- a sliding window containing the 50 most recent previous negative readings, while for TooManyFiringCategories we retuned every 10 readings.

Table 7.3 shows the results we obtained. Every aggregator exhibited a very low FNR, in line with those of test with locked state: hence, we can say that retuning does not affect the capability of evaluated techniques to detect attacks. Moreover, as expected, all techniques exhibited a sensibly lower FPR, with the only exceptions of LOF and Gaussian Parzen, whose average and, especially, maximum FPR are still pretty high.

In summary, all the techniques, with exception of LOF and Gaussian Parzen, are able to correctly detect almost all the simulated attacks while not raising too many false alarms, for all pages. More in detail, LOF and Gaussian Parzen showed too many false positives on the page Java – Top 25 bugs.

7.2.5 Discussion

According to our experimental evaluation, almost all techniques (with the exception of LOF and Gaussian Parzen) show results, in terms of FNR and FPR, which are sufficiently low to deserve further consideration. In particular, most techniques achieve an average

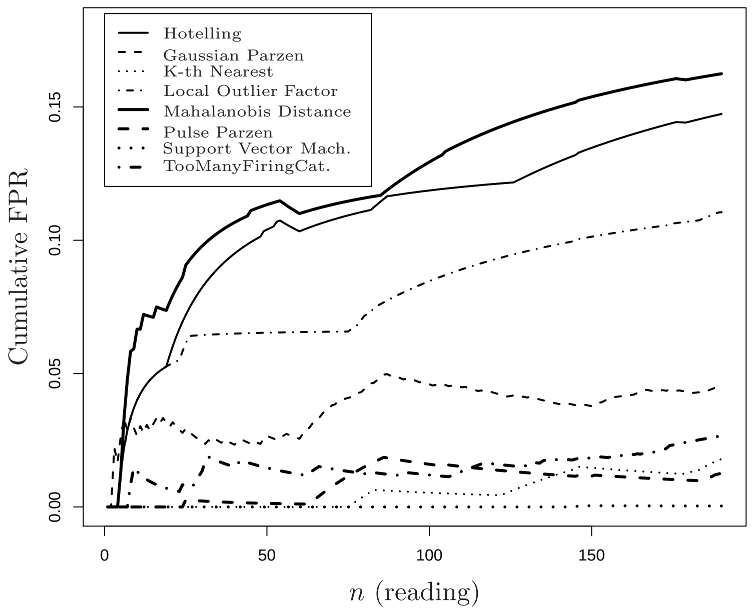


Figure 7.1: Cumulative False Positive ratio

Table 7.3: Long term effectiveness with retuning.

	FPR (%)			FNR (%)		
Aggregator	avg	max	min	avg	max	min
TooManyFiringCategories	0.2	1.0	0.0	0.0	0.0	0.0
K-th Nearest	0.2	1.6	0.0	0.1	1.3	0.0
Local Outlier Factor	1.4	10.0	0.0	0.1	1.3	0.0
Hotelling	0.5	2.1	0.0	0.1	1.3	0.0
Mahalanobis	0.7	2.6	0.0	0.2	1.3	0.0
Parzen Pulse	0.4	5.8	0.0	0.0	0.0	0.0
Parzen Gaussian	2.6	37.9	0.0	0.1	1.3	0.0
Support Vector Machines	0.1	0.5	0.0	0.0	0.0	0.0

FPR lower than 1%, while being able to correctly detect almost all the simulated attacks (FNR $\simeq$ 0%). We remark that such a lower FPR is equivalent, in our scenario, to about 1 false positive raised every month for each page. Such a finding suggests that, with most of the proposed techniques, the realization of a large-scale monitoring service is a feasible solution (see also Section 4.9 for a deeper analysis about scalability of the proposed solution).

Broadly speaking, however, none of such techniques appears to deliver significant performance improvements with respect to our earlier TooManyFiringCategories proposal. Support Vector Machines are the most promising alternative from this point of view. Since that technique requires an archive of attacks, however, it may be useful to investigate more deeply the relation between quality of that archive and resulting performance.

Based on this analysis, we believe also that a domain knowledge-based detection algorithm benefits from two important advantages:

1. An intrinsic feature of the TooManyFiringCategories aggregator is that the knowledge about v elements meaning can be used by the framework to provide the human operator with meaningful indications in case of a positive. For example, the operator could be notified with some indication about an anomalous number of links in the page or about a tag that was not present in the page despite being supposed to be. Such indications can hardly be provided using the other techniques.

2. The TooManyFiringCategories aggregator does not require a feature selection. While increasing performance for the techniques we tested, thus definitely making defacement detection effective with them, feature selection introduces more opportunities for attacks that remain hidden within the analyzed profile. Any attack affecting only elements of v that are not taken into account after the feature selection, cannot be detected by a detection algorithm that requires feature selection. The practical relevance of this issue (i.e., which attacks indeed fall in this category) certainly deserves further analysis.

Chapter 8

Large-scale study on reaction time to web site defacements

Statistics available about defacements are primarily concerned with the *number* of such incidents [83, 72]. Besides, an abundant literature about first-class attacks is available: interesting defacements occurred recently include, e.g., those suffered by the Company Registration Office in Ireland, which was defaced in December 2006 and remained so until mid-January 2007 [69]; by the Italian Air Force, which was defaced on January 26th 2007 by a Turkish attacker protesting against "who is supporting the war"; by the United Nations [30] and by the Spanish Ministry of Housing [25] in August 2007. To the best of our knowledge, there are no statistics available regarding the typical *duration* of a defacement. This information is crucial for improving the understanding of this phenomenon: a defacement lasting a few weeks is clearly much more harmful than one lasting a few minutes. Indeed, the defacements mentioned above lasted for a time ranging from a few hours to several weeks.

In this chapter we analyze the actual reaction time to defacements. We base our analysis on a sample of more than 62000 real incidents that we monitored in near real time for approximately two months. We were quite surprised by the slowness of the typical reaction, which is in the order of several days. For example, about 43% of the defacements in our sample lasted for at least one week and more than 37% of the defacements was still in place after two weeks.

To extract as much useful information as possible from our results, we analyzed the reaction time by isolating the *mass* defacements, i.e., defacements at different sites that are associated with the same IP address and should hence be hosted by a professionally administered infrastructure. We expected to observe for mass defacements a significantly faster reaction time, but this expectation was confirmed only in minor part. We also analyzed the reaction time as a function of the PageRank value of the corresponding pages, which is a form of measuring the "importance" of the page [56]. We found that pages with higher PageRank value indeed tend to exhibit a prompter reaction. Yet, the reaction time tends to be unacceptably high even in pages whose importance, as quantified by the PageRank, is manifest.

Our results are clearly to be interpreted with care—for example, we have no data about the security budget at the organizations involved, nor about the number of clients that actually visited the defaced sites. Despite their intrinsic limitations, though, we believe that our findings may greatly improve the understanding of this phenomenon and highlight issues deserving attention by the research community. Besides, any large-scale study of this kind will suffer from similar obstacles: we do not see any sound alternative for obtaining data significantly more accurate than ours.

8.1 Overview of our methodology

The basic idea behind our methodology is rather simple. We take advantage of Zone-H, a public web-based archive devoted to collecting evidence of defacements (http://www.zone-h.org). When a hacker has defaced a web page, the hacker himself—or any other user—may notify Zone-H about the URL of the page attacked. The staff at Zone-H fetches the page from the web and verifies whether it indeed exhibits evidence of a defacement, in which case the page is stored within the Zone-H archive. Zone-H is used widely—more than 490000 defacements have been stored in year 2005—thus we decided to base our study on this archive.

We constructed a tool composed by two components:

- a *crawler*, that fetches every hour from Zone-H the list of pages inserted into the archive;
- a *checker*, that compares every hour the content obtained by the crawler to the content exposed on the web.

More in detail, let u denote an URL obtained by the crawler, let $Z(u)$ denote the corresponding content stored at Zone-H and let $W(u)$ denote the content of URL u obtained by the checker from the web. Essentially, we consider that the defacement has been detected when $Z(u) \neq W(u)$. As clarified in more detail later, we approximate the time length of this incident with the interval starting from when the defacement was applied to when the inequality first holds.

The implementation of this basic idea is quite more complex than it appears, as will become evident from the description below. For example, we have observed defaced pages including dynamic content, in which case $W(u)$ differs from $Z(u)$ even though the page has not been healed. We have observed defaced pages with intermittent connectivity, or that are never reachable. We have also observed pages that have been healed before our first check. Extracting useful information from all possible cases automatically turned out to be a significant challenge, also because the number of sites that we needed to check every hour grew somewhat beyond our expectations: we usually had to check more than 20000 pages every hour.

Defacements can be subdivided in two broad categories: *substitutive* defacements—i.e., those replacing legitimate content—and *additive* defacements—i.e., those adding a web page at an URL where there should not be any page. Additive defacements are usually not observed by visitors of the attacked site and are meant to constitute a sort of hidden proof of the successful attack. These defacements could become particularly harmful as part of a more complex attack strategy, though. For example, a phishing attack could direct users to a fake login page inserted *within* a site trusted by its users: this is exactly what happened in early 2008 to the site of an Italian bank [53]. The Zone-H archive does not provide any indication as to whether a given defacement is substitutive or additive. As will be clear in the next section, this fact has complicated our analysis further.

8.2 Methodology

The Zone-H Digital Attacks Archive contains a list of URLs corresponding to defacements which have been validated by the Zone-H staff and is maintained as follows (see Figure 8.1(a)):

1. the attacker notifies the Zone-H staff about the URL u of a defaced page (via a web form) and Zone-H downloads automatically a snapshot $Z(u)$ of the claimed defacement;

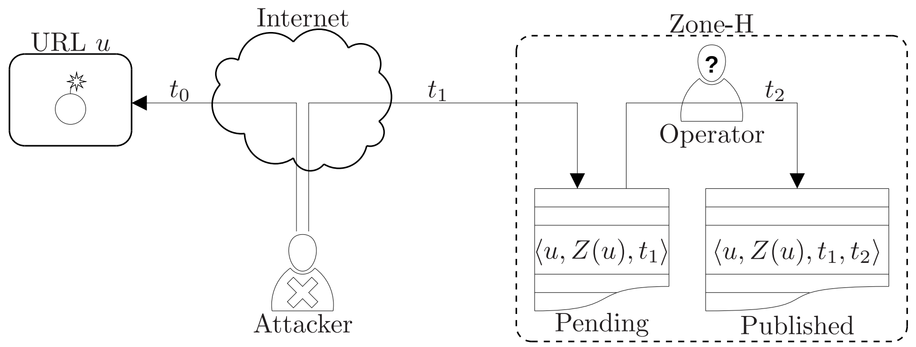
2. some time later, a human operator at Zone-H verifies the snapshot and, provided the notification is deemed valid, inserts in the Zone-H archive an entry $\langle u, Z(u), t_1, t_2\rangle$, where t_1 denotes the time instant at which the snapshot $Z(u)$ was taken and t_2 denotes the time instant at which the defacement has been verified.

The human operator observes only the snapshot $Z(u)$, which means that at t_2 the page may or may not be still defaced—the administrator at u might have healed u very quickly.

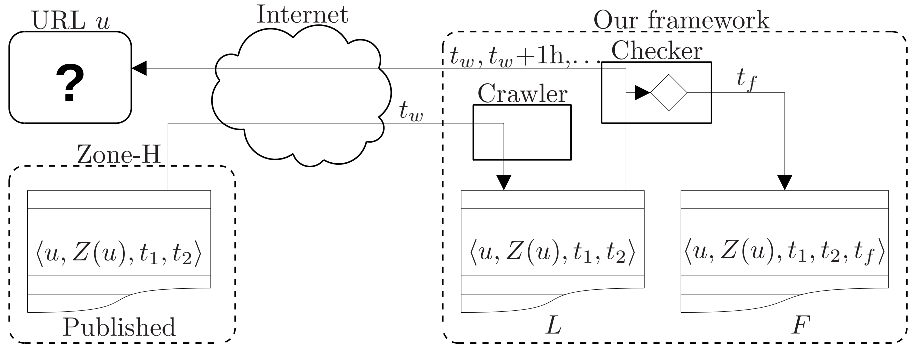
Our *crawler* builds a list L of defaced pages by querying every hour the Zone-H archive. The list was initialized with the entries inserted on the day we started the experiment. The crawler then inserts in L each new entry found at Zone-H. For obvious efficiency reasons, $Z(u)$ is actually stored in the form of a hash and all comparisons to the page u as exposed on the web, i.e., $W(u)$, are made by comparing the respective hashes. We shall not mention this issue any further.

Our *checker* checks each entry of L every hour. Actually, the checker starts its execution every hour and must complete all checks before the next run—the time between two consecutive checks of the same entry, thus, is not exactly one hour. The checker implements, for each entry of L, a finite state machine aimed at tracking the status of the defacement—e.g., whether it is still in place, it has been removed, the page is not reachable and so on. We provide only an overview of this state machine for ease of presentation: we omit all state transition rules and focus only on the meaning of the final states, which are those containing the crucial information for our analysis.

At each checking step the checker fetches $W(u)$ from the web and compares it to $Z(u)$ as stored in L. The state machine receives an input describing the outcome of the check and performs a state transition. The outcome of the check is one of the following:



(a) Zone-H operations


URL u
Internet
Our framework
t_w, t_w+1h,...
Checker
t_f
Crawler
t_w
Zone-H
$\langle u, Z(u), t_1, t_2 \rangle$
$\langle u, Z(u), t_1, t_2 \rangle$
$\langle u, Z(u), t_1, t_2, t_f \rangle$
Published
L
F

(b) Our framework operations

Figure 8.1: Operations performed by Zone-H (above) and our framework (below). Each arrow corresponds to an action and is annotated with the time instant at which the action is performed (see the text).

- Unreachable, meaning that the server did not reply within 20 seconds or it replied with an HTTP error (i.e., 404 Not Found, 503 Service Unavailable, and so on);
- Equal, meaning that $Z(u)$ and $W(u)$ are identical;
- NotEqual, otherwise.

Whenever the state machine reaches one of the final states, described below, the checker removes the corresponding entry from L and inserts it into another list F. Each entry in F includes an indication of the final state and of the instant t_f in which this state was reached. All our results are based on the content of F. Our monitoring framework is summarized in Figure 8.1(b), where t_w denotes the time instant at which we fetched $W(u)$ for the first time.

We associate each entry in L with the time instant t_f at which the entry reached the final state. We say that we could *verify* a defacement for u to mean that our checker indeed verified at least once that $Z(u) = W(u)$. We define $T_{\max} = 2$ weeks. All the possible final states are listed below along with their meaning:

Undetected We verified the defacement and the defacement remained in place for at least $T_{\max}$. An entry enters this final state after a sequence of verifications spanning a time interval $T_{\max}$, without any interruption. Sites that were occasionally unreachable, thus, were observed for more than $T_{\max}$.

Removed We verified the defacement but later the page remained systematically unreachable for at least 1 week. Our domain knowledge suggests us that entries in this final state correspond to additive defacements which have been removed. We set t_f to the instant of the first unreachable observation among the consecutive ones lasting 1 week.

Patched We verified the defacement and the page changed sometime later. This final state may be due to any of the following reasons:

A. the legitimate content has been restored;

B. a maintenance or error message replaced the defacement;

C. the defacement was *dynamic* or it was a partial defacement of a dynamic web page—thereby leading to different hash values at different checks;

D. another defacement replaced the first defacement.

We could not discriminate among these cases automatically, because we do not know which was the legitimate content. We visually inspected a sample of 200 entries in this final state and found that 65%, 20%, 3%, 8% of them concerned cases A, B, C and D respectively; we were not able to verify the remaining 4%. In other words, 85% of this sample identify a form of reaction to the defacement (cases A and B).

NotVerified We did *not* verify the defacement. In other words, the first snapshot $W(u)$ did not satisfy $W(u) = Z(u)$. Our domain knowledge suggests that there may be two main reasons for this outcome:

A. before our first check, either the original content were restored or a maintenance message had replaced the defacement;

B. the defacement was dynamic or it was a partial defacement of a dynamic web page.

We visually inspected a sample of 200 entries in this final state and found that 16% of them were in condition A, 83% were in condition B; we were not able to verify the remaining 1%.

NotChecked We could not even fetch any snapshot of the page. That is, we could not fetch $W(u)$ for $T_{\max}$. The most likely reason for this final state is that the entry represents an additive defacement that was removed before our first check.

Table 8.1: Summary of final states.

State	# of entries	Perc.
NotVerified	18608	29.8%
NotChecked	2823	4.5%
Removed	7437	11.9%
Patched	16593	26.6%
Undetected	16972	27.2%
Total	62433	100.0%

Based on the meaning of final states we constructed estimates for the incident reaction time T_r as follows. For Undetected entries, $T_r \geq T_{\max}$. For entries either Patched or Removed, $T_r \approx t_f - t_1$. For entries either NotVerified or NotChecked, all that we can safely say is that, if the page has been indeed healed, it has been healed at some instant before our first check.

We attempted to smooth the effects of this uncertainty on our analysis by discussing our results in two ways: first, by taking into account only entries that were either Undetected/Patched/Removed (for simplicity, Verified entries); then, by making two optimistic hypotheses about the reaction time of NotVerified/NotChecked entries.

8.3 Results

We started the crawler and the checker on February 17th 2007. We stopped the crawler on April 13th 2007, after 49 days. The checker continued to run until L was empty, which took further 13 days.

Table 8.1 summarizes our results in terms of final states. The salient observation is that we found more than 62000 new defacements on Zone-H in 49 days, which corresponds to about 1250 new URLs every day—a figure sufficiently large to make the practical relevance of this phenomenon evident. Table 8.1 shows that there are 41002 Verified entries, accounting for 65.7% of all entries.

8.3.1 Analysis for Verified entries

Figure 8.2 shows the number of Verified entries (y-axis) that have been detected within a given reaction time t (x-axis)—i.e., entries for which $T_r \leq t$. The secondary y-axis expresses the number of detected entries as a percentage of all Verified entries. The two dotted lines highlight the t values corresponding to 1 day and 1 week: a reaction occurs within one day only in less than 25% of the Verified entries; a reaction occurs within 1 week in about 50% of them. The average reaction time is $\overline{T_r} = 72.4$ hours. We were quite surprised by such long reaction times and such long-tailed distributions, which appear to be unacceptably long under any metric.

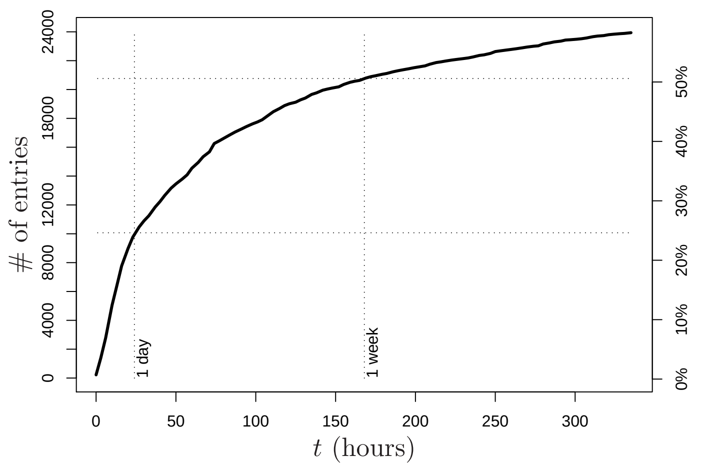


Figure 8.2: Number of Verified entries (y-axis) that have been detected within a given t (x-axis); two vertical dotted lines are traced at values of t corresponding to 1 day and 1 week. The secondary (right) y-axis expresses the number of detected entries as a percentage of all Verified entries.

Figure 8.3 characterizes the incident reaction time T_r for Patched and Removed entries separately (recall that for Undetected entries we assumed, optimistically, that $T_r = T_{\max} = 2$ weeks): the x-axis shows the time in days and the y-axis shows the number of entries as a percentage of the respective state. For example, $T_r \leq 24$ (hours) for about 47% of Patched entries, $24 < T_r \leq 48$ for about 14% of Patched entries, and so on. The figure clearly suggests that Patched entries are generally detected earlier than Removed entries. This observation is confirmed by the average reaction time $\overline{T_r}$, which is 66.1 hours for the former and 86.3 hours for the latter. We believe this difference is due to the fact that most of Patched entries correspond to substitutive defacements whereas most of Removed entries correspond to additive defacements, that are intrinsically more difficult to detect.

We remark that the above discussion is based upon the optimistic assumption that all Patched entries—roughly 40% of all Verified entries—always involve a reaction to a defacement. Instead, a visual inspection on a sample of 200 Patched entries (see the definition of the Patched final state) suggested us that only 85% of Patched entries indeed identify a form of reaction to the attack. It follows that the actual distribution of reaction time is worse than shown here: if we estimate the reaction time by assuming that only 85% of all Patched entries retain their T_r while the remaining 15% are considered as Undetected, we find a sensible loss in the promptness of detection: first-day detection rate for all Verified entries becomes 21.7% (vs. 24.6%) and 2nd-week detection rate becomes 52.4% (vs. 58.4%).

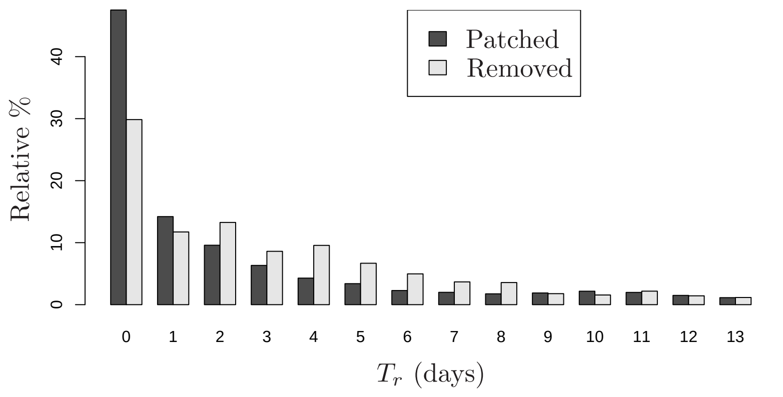


Figure 8.3: Distribution of incident reaction time T_r for Patched and Removed entries.

8.3.2 Analysis for mass defacements of Verified entries

We performed a further analysis by distinguishing between *mass* defacements and *single* defacements. A mass defacement consists in defacing multiple URLs that are associated with the same IP address. This phenomenon typically occurs when an attacker manages to exploit a vulnerability in a server hosting multiple web sites. We thought that the reaction to a mass defacement should be fast because, broadly speaking, sites involved in a mass defacement should be part of a professionally administered hosting infrastructure.

We define an entry as a mass defacement if there is at least another entry with the same tuple $\langle \texttt{IP address}, Z(u), t_1 \rangle$. Table 8.2 summarizes the results. The first two columns describe the size of each subset: mass defacements and single defacements account for 72.5% and 27.5% of the Verified entries, respectively. The remaining columns indicate the reaction time for each subset. For example, 28.8% of mass defacements have a reaction within 1 day. The bottom row indicates the baseline, i.e., the corresponding values computed across the whole set of Verified entries.

There is indeed a sensible gap between the reaction time of the two subsets, both in average and percentage at the salient instants considered. This fact confirms our expectation than the reaction to mass defacements occurs more quickly than to single defacements. Nevertheless, the reaction time remains quite high even in this case: after 1 week only half of the entries is detected and the average reaction time is 2.7 days. It seems reasonable to claim that defacements and reaction to defacements are a significant issue also at web site providers.

8.3.3 Analysis of Verified entries based on the respective PageRank value

We attempted to gain some insights into the possible correlation between the "importance" of a defaced web page and the corresponding reaction time. Quantifying the former is obviously quite hard. For example, a web page could be useful only to a very small set of users, which however could depend heavily on the integrity of that page. Or, a web page could remain defaced only during a time interval in which few of its users, if

Table 8.2: T_r characterization for mass and single defacements separately.

Partition	# of entries	% %	Det. at 1 day	Det. at 1 week	Det. 2 weeks	$\overline{T_r}$ (hours)
Mass	29719	72.5%	28.8%	53.2%	59.9%	64.9
Single	11283	27.5%	13.3%	43.9%	54.4%	94.2
Total	*41002*	*100.0%*	*24.6%*	*50.6%*	*58.4%*	*72.4*

any, accessed that page.

Rather than plainly neglecting this issue, however, we decided to incorporate in our analysis the PageRank values of the defaced pages [56]. The PageRank of a page is an integer in the range $[-1, 10]$: the higher the value, the more "important" the page (see the cited paper for details about the metric); value -1 means that a PageRank value is not available for the corresponding URL. We chose this index for convenience:

- we could collect the PageRank value for each entry considered in our analysis by querying the Google Toolbar in an automated way;
- we could not see any sensible alternative for quantifying the importance of a page in a study of this kind—tens of thousands of defaced pages spread around the world and whose access pattern is unknown.

The corresponding results must clearly be interpreted carefully, as we cannot establish any direct relation between the importance of a page, as quantified by its PageRank, and the actual damage caused by the defacement. For example, pages with a very small PageRank value may actually be, and often are indeed, quite important to their users.

For each entry with URL u, we collected both the PageRank value p_{page} associated with u, and the PageRank value p_{domain} associated with the domain of u. Figure 8.4 shows the distribution of p_{page} and p_{domain} across all the entries. It can be seen that p_{page} is -1 for approximately 39000 entries, but there is a substantial number of entries for which p_{page} is not negligible: for 5488 entries—roughly 9% of all entries—p_{page} is 3 or above. Similar remarks can be made concerning p_{domain}: there is a large number of entries for which $p_{\text{domain}} = -1$, but there are also thousands of entries for which p_{domain} is 3 or above. These results suggest that defacements do affect also PageRank-important sites.

Table 8.3 focuses on Verified entries: it shows number and percentage of Removed, Patched and Undetected entries for which $p_{\text{page}} \neq -1$. It can be seen that the relative occurrence of entries with $p_{\text{page}} \neq -1$ is much higher in Patched entries than in Removed entries. This fact confirms the intuition that most of Removed entries probably correspond to additive defacements, i.e., pages that are not supposed to exist—for such pages the PageRank value is not available.

Figure 8.5 correlates the reaction time to the PageRank and constitutes the salient point of this analysis: it shows the percentage of Verified entries that have been detected

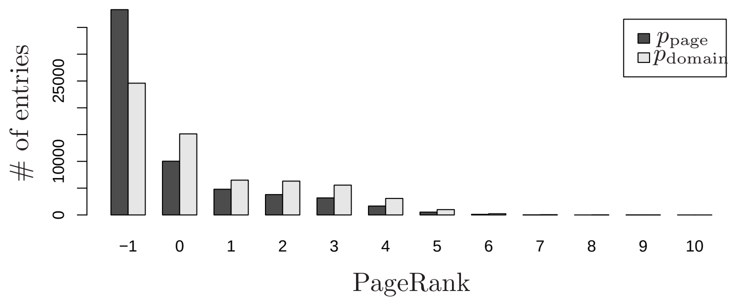


Figure 8.4: Distribution of p_{page} and p_{domain} for all considered entries: value -1 corresponds to URLs for which PageRank is not available.

Table 8.3: Availability of PageRank value for Verified entries.

	# of entries		%
State	$p_{\text{page}} \geq 0$	$p_{\text{page}} = -1$	$p_{\text{page}} \geq 0$
Removed	591	7437	7.4%
Patched	8930	16593	35.0%
Undetected	3739	16972	18.1%

within one day or one week, as a function of their p_{page} values. For example, in 35.3% of the entries with $p_{\text{page}} = 0$ some form of reaction (either patching or removing) occurs within 1 day. As a baseline, dotted horizontal lines represents the values averaged for all Verified entries, independently of their PageRank value (Section 8.3.1).

The figure confirms the intuition that reaction to defacement of pages with a higher PageRank value is indeed faster. This effect is more prominent for detection percentages at one week. It seems fair to claim, though, that even from this point of view the reaction time is not as fast as one would probably expect—we never observed a 1-day detection percentage significantly greater than 50%, for example.

8.3.4 Analysis for all entries

We wanted to verify whether incident reaction times presented in the previous sections were distorted by the fact that we considered only Verified entries, hence discarding the remaining 34.3% of the 62433 entries. To this end, we reasoned about how to take into account entries that were either NotVerified (we could not observe that the site appeared as stored in Zone-H) or NotChecked (we could not even fetch anything from the site for at least two weeks).

We remark that the uncertainty in the reaction time of Verified entries is in the order of a few tens of minutes, because the checker runs every hour. Such uncertainty is unfortunately much larger for entries that are either NotVerified or NotChecked, as we do

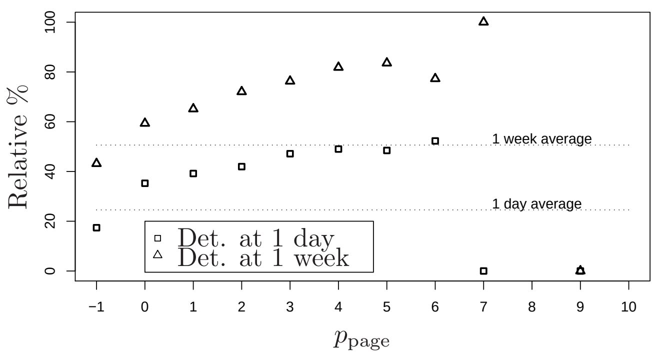


Figure 8.5: Relative percentage of Verified entries (y-axis) that have been detected within one day (square) or one week (triangle), as function of the corresponding p_{page} (x-axis). Two horizontal dotted lines are traced at values of % corresponding to the 1 day and 1 week averages computed for all Verified entries. Percentages for $p_{\text{page}} > 6$ are not significant, since there are very few entries with such a PageRank value.

not even know for sure whether these entries were indeed healed (recall, for example, that a NotVerified entries could correspond to a dynamic defacement or to a defacement that affected only static portions of a page). Any reasoning about NotVerified/NotChecked entries, thus, cannot avoid to suffer from such uncertainty.

We formulated two optimistic hypotheses about the reaction time for NotVerified/NotChecked entries and used these hypotheses for finding an upper bound to the curve in Figure 8.2:

Hypothesis A A very optimistic scenario in which we assumed that each NotVerified/NotChecked entry is associated with $T_r = 0$—i.e., a reaction to the defacement at u occurred as soon as Zone-H downloaded $Z(u)$ at t_1.

Hypothesis B A slightly less optimistic scenario in which NotChecked entries are handled as above but we assume that $T_r = 0$ only for 16% of the NotVerified entries, consistently with our visual inspection of a sample (Section 8.2). Concerning the remaining 84% of NotVerified entries, we assumed the same results that we observed for Verified entries: we considered as Undetected a portion consistent with the relative occurrence of Undetected entries with respect to Verified entries (41.4%); we assumed a reaction time T_r whose distribution is the same as that of Patched and Removed entries (Figure 8.2) for the remaining portion.

Figure 8.6 presents the corresponding results. As expected, both curves show a significant number of entries detected immediately ($T_r \leq t = 0$) and stay above the curve for the Verified entries. In scenario (A) the first-day and first-week detection rates are 50.5% and 67.6%, as opposed to 24.6% and 50.6% respectively. The improvement is significant, yet it seems reasonable to claim that the reaction time remains unacceptably long. Concerning the slightly less optimistic scenario (B), the improvement is much

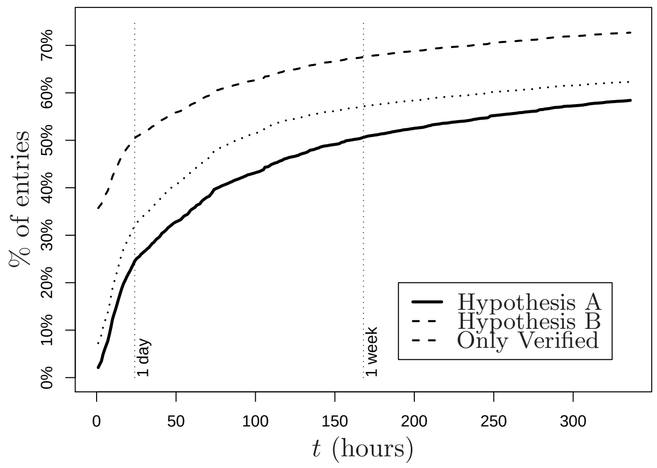


Figure 8.6: Percentage of entries (y-axis) that have been detected within a given t (x-axis) obtained with two different optimistic hypotheses about NotVerified/NotChecked entries; two vertical dotted lines are traced at values of t corresponding to 1 day and 1 week.

smaller: 7.5 and 6.6 percentage points respectively. Average reaction time are 38.3 and 54.5 hours for scenarios (A) and (B) respectively, in respect to 72.4 hours for the Verified entries.

Bibliography

[1] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, and C. D. Spyropoulos. An experimental comparison of naive Bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 160–167, New York, NY, USA, 2000. ACM Press. [cited at p. 12]

[2] A. Anitha and V. Vaidehi. Context based Application Level Intrusion Detection System. In *ICNS '06: Proceedings of the International conference on Networking and Services*, page 16, Los Alamitos, CA, USA, 2006. IEEE Computer Society. [cited at p. 10]

[3] G. K. Baah, A. Gray, and M. J. Harrold. On-line anomaly detection of deployed software: a statistical machine learning approach. In *SOQUA '06: Proceedings of the 3rd International Workshop on Software Quality Assurance*, pages 70–77, New York, NY, USA, 2006. ACM Press. [cited at p. 52]

[4] M. Banikazemi, D. Poff, and B. Abali. Storage-based file system integrity checker. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 57–63, New York, NY, USA, 2005. ACM Press. [cited at p. 8]

[5] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Annual Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh, Pennsylvania, United States, 1992. ACM. [cited at p. 75, 78]

[6] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. *SIGMOD Rec*, 29:93–104, 2000. [cited at p. 75, 77]

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, 1997. [cited at p. 9]

[8] C. E. Brodley and M. A. Friedl. Identifying Mislabeled Training Data. *J. Artif. Intell. Res. (JAIR)*, 11:131–167, 1999. [cited at p. 53]

[9] C. C. Chang and C. J. Lin. Libsvm: a library for support vector machines. Technical report, 2001. Software available at http://www.csie.ntu.edu.tw/cjlin/libsvm. [cited at p. 78]

[10] H.-Y. Chang, S. F. Wu, and Y. F. Jou. Real-time protocol analysis for detecting link-state routing protocol attacks. *ACM Trans. Inf. Syst. Secur.*, 4(1):1–36, 2001. [cited at p. 10]

[11] S. N. Chari and P.-C. Cheng. BlueBoX: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003. [cited at p. 10]

[12] Y. Chen, A. Abraham, and B. Yang. Hybrid flexible neural-tree-based intrusion detection systems. *International Journal of Intelligent Systems*, 22(4):337–352, 2007. [cited at p. 66]

[13] T. M. Chilimbi and V. Ganapathy. HeapMD: identifying heap-based bugs using anomaly detection. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 219–228, New York, NY, USA, 2006. ACM Press. [cited at p. 10]

[14] L. F. Cranor and B. A. LaMacchia. Spam! *Commun. ACM*, 41(8):74–83, 1998. [cited at p. 12]

[15] D. Dasey. Cyber threat to personal details. Technical report, The Sydney Morning Herald, Oct. 2007. Available at http://www.smh.com.au/news/technology/cyber-threat-to-personal-details/2007/10/13/1191696235979.html. [cited at p. 3]

[16] D. E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, 1987. [cited at p. 2, 10, 42, 75]

[17] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener. A large-scale study of the evolution of web pages. *Softw. Pract. Exper.*, 34(2):213–237, 2004. [cited at p. 9]

[18] W. Fone and P. Gregory. Web page defacement countermeasures. In *Proceedings of the 3rd International Symposium on Communication Systems Networks and Digital Signal Processing*, pages 26–29, Newcastle, UK, July 2002. IEE/IEEE/BCS. [cited at p. 7]

[19] G. Forman and I. Cohen. Learning from little: comparison of classifiers given little training. In *PKDD '04: Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 161–172, New York, NY, USA, 2004. Springer-Verlag New York, Inc. [cited at p. 53]

[20] A. Y. Fu, L. Wenyin, and X. Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd). *IEEE Trans. Dependable Secur. Comput.*, 3(4):301–311, 2006. [cited at p. 8]

[21] A. Gehani, S. Chandra, and G. Kedem. Augmenting storage with an intrusion response primitive to ensure the security of critical data. In *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 114–124, New York, NY, USA, 2006. ACM Press. [cited at p. 8]

[22] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2006 CSI/FBI Computer Crime and Security Survey. Security survey, Computer Security Institute, 2006. [cited at p. 3]

[23] A. K. Gosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *ACSAC '98: Proceedings of the 14th Annual Computer Security Applications Conference*, page 259, Los Alamitos, CA, USA, 1998. IEEE Computer Society. [cited at p. 10, 75]

[24] HA. Madagascar Mass Government Defacement. Technical report, Hacker's Attacks - Web Defacements Blog, Oct. 2007. Available at http://calima.serapis.net/blogs/index.php?/archives/116-Madagascar-Mass-Government-Defacement.html. [cited at p. 3]

[25] HB. Hacker attacks the Ministry for Housing website as Spanish mortgages come under the international spotlight. Technical report, Typically Spanish, Aug. 2007. Available at http://www.typicallyspanish.com/news/publish/article_12212.shtml. [cited at p. 3, 85]

[26] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A Network Security Monitor. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 296, Los Alamitos, CA, USA, 1990. IEEE Computer Society. [cited at p. 10]

[27] V. Hodge and J. Austin. A Survey of Outlier Detection Methodologies. *Artif. Intell. Rev.*, 22(2):85–126, 2004. [cited at p. 53]

[28] H. Hotelling. The generalization of student's ratio. *The Annals of Mathematical Statistics*, 2:360–378, 1931. [cited at p. 75, 77]

[29] W. Hu, Y. Liao, and V. R. Vemuri. Robust Support Vector Machines for Anomaly Detection in Computer Security. In *ICMLA*, pages 168–174, 2003. [cited at p. 53]

[30] G. Keizer. 'Hackers' deface UN site. Technical report, ComputerWorld Security, Aug. 2007. Available at http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9030318. [cited at p. 3, 85]

[31] T. Kemp. Security's Shaky State, 2005. Available at http://www.informationweek.com/industries/showArticle.jhtml?articleID=174900279. [cited at p. 2]

[32] E. Kim and S. Kim. Anomaly detection in network security based on nonparametric techniques. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–2, 2006. [cited at p. 76, 78]

[33] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM Press. [cited at p. 8]

[34] J. Kirk. Microsoft's U.K. Web site hit by SQL injection attack. Technical report, ComputerWorld Security, Jun. 2007. Available at http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9025941. [cited at p. 3]

[35] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, December 1992. [cited at p. 65]

[36] C. Kruegel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 201–208, New York, NY, USA, 2002. ACM Press. [cited at p. 10, 75]

[37] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261, New York, NY, USA, 2003. ACM Press. [cited at p. 10, 11, 52, 75]

[38] T. Lane and C. E. Brodley. An application of machine learning to anomaly detection. In *Proceedings of the Twentieth National Information Systems Security Conference*, volume 1, pages 366–380, Gaithersburg, MD, 1997. The National Institute of Standards and Technology and the National Computer Security Center, National Institute of Standards and Technology. [cited at p. 52]

[39] T. D. Lane. *Machine learning techniques for the computer security domain of anomaly detection.* PhD thesis, Purdue University, 2000. Major Professor-Carla E. Brodley. [cited at p. 52]

[40] P. Laskov, C. Schäfer, and I. V. Kotenko. Intrusion detection in unlabeled data with quarter-sphere Support Vector Machines. In *DIMVA*, pages 71–82, 2004. [cited at p. 54]

[41] A. Lazarevic, L. Ertöz, V. Kumar, A. Ozgur, and J. Srivastava. A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection. In *Proceedings of the Third SIAM International Conference on Data Mining*, San Francisco, CA, 2003. SIAM. [cited at p. 11, 31, 49, 76, 77, 78]

[42] K. Li and G. Teng. Unsupervised svm based on p-kernels for anomaly detection. *First International Conference on Innovative Computing, Information and Control - Volume II (ICICIC'06)*, 2:59–62, 2006. [cited at p. 52]

[43] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation. In *RAID '00:*

Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection, pages 162–182, London, UK, 2000. Springer-Verlag. [cited at p. 31]

[44] W. Liu, X. Deng, G. Huang, and A. Y. Fu. An Antiphishing Strategy Based on Visual Similarity Assessment. *IEEE Internet Computing*, 10(2):58–65, 2006. [cited at p. 8]

[45] P. C. Mahalanobis. On the generalized distance in statistics. In *Proceedings of the National Institute of Science of India*, volume 12, pages 49–55, 1936. [cited at p. 75, 77]

[46] M. Mahoney and P. Chan. Phad: Packet header anomaly detection for identifying hostile network traffic. Technical report, Florida Tech., CS-2001-4, 2001. [cited at p. 53]

[47] 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. Inf. Syst. Secur.*, 3(4):262–294, 2000. [cited at p. 31]

[48] R. McMillan. Bad things lurking on government sites. Technical report, InfoWorld, Oct. 2007. Available at http://www.infoworld.com/article/07/10/04/Bad-things-lurking-on-government-sites_1.html. [cited at p. 3, 49]

[49] C. C. Michael and A. Ghosh. Simple, state-based approaches to program-based anomaly detection. *ACM Trans. Inf. Syst. Secur.*, 5(3):203–237, 2002. [cited at p. 10, 49]

[50] G. Mishne, D. Carmel, and R. Lempel. Blocking Blog Spam with Language Model Disagreement. In *1st International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, Chiba, Japan, May 2005. AIRWeb. [cited at p. 12]

[51] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1702–1707, 2002. [cited at p. 78]

[52] S. Mukkamala, A. H. Sung, and A. Abraham. Modeling intrusion detection systems using linear genetic programming approach. In *IEA/AIE'2004: Proceedings of the 17th international conference on Innovations in applied artificial intelligence*, pages 633–642. Springer Springer Verlag Inc, 2004. [cited at p. 66]

[53] P. Mutton. Italian Bank's XSS Opportunity Seized by Fraudsters. Technical report, Netcraft, Jan. 2008. Available at http://news.netcraft.com/archives/2008/01/08/italian_banks_xss_opportunity_seized_by_fraudsters.html. [cited at p. 87]

[54] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006. [cited at p. 10, 11, 49, 55, 75]

[55] A. Ntoulas, J. Cho, and C. Olston. What's New on the Web? The Evolution of the Web from a Search Engine Perspective. In *Proceedings of the 13th International World Wide Web Conference*, pages 1–12, New York, NY, USA, May 2004. ACM Press. [cited at p. 9, 14]

[56] L. Page, S. Brin, M. Rajeev, and W. Terry. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998. [cited at p. 12, 85, 93]

[57] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33:1065–1076, Sept. 1962. [cited at p. 75, 78]

[58] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Networks*, 51(12):3448–3470, 2007. [cited at p. 11]

[59] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. Soules, G. R. Goodson, and G. R. Ganger. Storage-Based Intrusion Detection: Watching Storage Activity for Suspicious Behavior. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC., Aug. 2003. USENIX. [cited at p. 8]

[60] D. Pulliam. Hackers deface Federal Executive Board Web sites. Technical report, Government Executive, Aug. 2006. Available at http://www.govexec.com/story_page.cfm?articleid=34812. [cited at p. 3]

[61] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. *SIGMOD Rec*, 29:427–438, 2000. [cited at p. 75, 76]

[62] R. Richardson. 2007 CSI Computer Crime and Security Survey. Security survey, Computer Security Institute, 2007. [cited at p. 3]

[63] A. Sanka, S. Chamakura, and S. Chakravarthy. A dataflow approach to efficient change detection of HTML/XML documents in WebVigiL. *Comput. Networks*, 50(10):1547–1563, 2006. [cited at p. 9]

[64] S. Sedaghat, J. Pieprzyk, and E. Vossough. On-the-fly web content integrity check boosts users' confidence. *Commun. ACM*, 45(11):33–37, 2002. [cited at p. 7]

[65] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society. [cited at p. 10]

[66] J. Shavlik and M. Shavlik. Selection, combination, and evaluation of effective software sensors for detecting abnormal computer usage. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 276–285, New York, NY, USA, 2004. ACM Press. [cited at p. 11, 49, 52]

[67] M.-L. Shyu, S.-C. Chen, K. Sarinnapakorn, and L. Chang. A Novel Anomaly Detection Scheme Based on Principal Component Classifier. In *Proceedings of the IEEE Foundations and New Directions of Data Mining Workshop, in conjunction with the Third IEEE International Conference on Data Mining (ICDM '03)*, pages 172–179, Melbourne, Florida, USA, 2003. IEEE. [cited at p. 11]

[68] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: techniques and applications. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36, New York, NY, USA, 2005. ACM Press. [cited at p. 8]

[69] G. Smith. CRO website hacked. Technical report, SiliconRepublic.com, 2007. Available at http://www.siliconrepublic.com/news/news.nv?storyid=single7819 and visited in April 30th 2007. [cited at p. 3, 85]

[70] G. Smith. CRO website hacked. Technical report, Silicon Republic, Feb. 2007. Available at http://www.siliconrepublic.com/news/news.nv?storyid=single7819. [cited at p. 3]

[71] D. Song, M. I. Heywood, and A. N. Zincir-Heywood. Training genetic programming on half a million patterns: an example from anomaly detection. *IEEE Trans. Evolutionary Computation*, 9(3):225–239, 2005. [cited at p. 66]

[72] K. N. Srijith. Analysis of the Defacement of Indian Web Sites. *First Monday*, 7(12), 2002. Available at http://firstmonday.org/issues/issue7_12/srijith/. [cited at p. 85]

[73] D. M. Tax and R. P. Duin. Data Domain Description using Support Vectors. In *ESANN*, pages 251–256, 1999. [cited at p. 54]

[74] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, pages 203–222, 2004. [cited at p. 54]

[75] T. Xia, G. Qu, S. Hariri, and M. Yousif. An efficient network intrusion detection method based on information theory and genetic algorithm. In *Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International*, pages 11–17, 2005. [cited at p. 66]

[76] N. Ye, Q. Chen, S. M. Emran, and S. Vilbert. Hotelling t2 multivariate profiling for anomaly detection. *Proc. 1st IEEE SMC Inform. Assurance and Security Workshop*, 2000. [cited at p. 77]

[77] N. Ye, S. M. Emran, Q. Chen, and S. Vilbert. Multivariate Statistical Analysis of Audit Trails for Host-Based Intrusion Detection. *IEEE Transactions on Computers*, 51(7):810–820, 2002. [cited at p. 11, 49]

[78] N. Ye, X. Li, Q. Chen, S. Emran, and M. Xu. Probabilistic techniques for intrusion detection based on computer audit data. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 31:266–274, 2001. [cited at p. 77]

[79] D.-Y. Yeung and C. Chow. Parzen-window network intrusion detectors. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 4, pages 385–388, 2002. [cited at p. 78]

[80] C. Yin, S. Tian, H. Huang, and J. He. Applying Genetic Programming to Evolve Learned Rules for Network Anomaly Detection. In *Advances in Natural Computation, First International Conference, ICNC 2005, Proceedings, Part III*, pages 323–331, 2005. [cited at p. 66]

[81] S. Zanero and S. M. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 412–419, New York, NY, USA, 2004. ACM Press. [cited at p. 10, 75]

[82] X. Zhu and X. Wu. Class noise vs. attribute noise: a quantitative study of their impacts. *Artif. Intell. Rev.*, 22(3):177–210, 2004. [cited at p. 52, 53]

[83] Zone-H.org. Statistics on Web Server Attacks for 2005. Technical report, The Internet Termometer, 2006. Available at http://www.zone-h.org/component/option,com_remository/Itemid,47/func,fileinfo/id,7771/. [cited at p. 1, 3, 4, 85]