A Framework for Large-Scale Detection of Web Site Defacements

10

ALBERTO BARTOLI, GIORGIO DAVANZO, and ERIC MEDVET
University of Trieste

Web site defacement, the process of introducing unauthorized modifications to a Web site, is a very common form of attack. In this paper we describe and evaluate experimentally a framework that may constitute the basis for a *defacement detection service* capable of monitoring thousands of remote Web sites systematically and automatically.

In our framework an organization may join the service by simply providing the URLs of the resources to be monitored along with the contact point of an administrator. The monitored organization may thus take advantage of the service with just a few mouse clicks, without installing any software locally or changing its own daily operational processes. Our approach is based on anomaly detection and allows monitoring the integrity of many remote Web resources automatically while remaining fully decoupled from them, in particular, without requiring any prior knowledge about those resources.

We evaluated our approach over a selection of dynamic resources and a set of publicly available defacements. The results are very satisfactory: all attacks are detected while keeping false positives to a minimum. We also assessed performance and scalability of our proposal and we found that it may indeed constitute the basis for actually deploying the proposed service on a large scale.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access (e.g., hacking, phreaking)*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network monitoring*

General Terms: Security

Additional Key Words and Phrases: Intrusion detection, monitoring service, Web site defacement, experimental evaluation

ACM Reference Format:
Bartoli, A., Davanzo, G., and Medvet, E. 2010. A framework for large-scale detection of Web site defacements. ACM Trans. Intern. Techn. 10, 3, Article 10 (October 2010), 37 pages.
DOI = 10.1145/1852096.1852098. http://doi.acm.org/10.1145/1852096.1852098.

Authors' addresses: A. Bartoli, G. Davanzo, and E. Medvet, DEEI, University of Trieste, Via Valerio, Trieste, Italy; email: bartoli.alberto@units.it, giorgio.davanzo@gmail.com, emedvet@units.it.

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 corrupts the content of a Web site may thus cause serious damage to the organization, as well as to its users or customers. Unfortunately, incidents of this sort are very common. Leaving aside the abundant and steadily expanding anectodal evidence, [Pulliam 2006; Kirk 2007; Smith 2007; McMillan 2007; Dasey 2007; Prefect 2010], it suffices to observe that more than 1.7 million Web sites were defaced between 2005 and 2007 and that the trend has been constantly growing in the recent years [Zone-H 2006; Gordon et al. 2006; Richardson 2007]. A defaced site typically contains only a few messages or images representing a sort of signature of the hacker that performed the defacement. It may also contain disturbing images or texts, political messages, and so on. Defacing a Web site may often be done quickly and cheaply, even in an automated manner [CERT/CC 2001]. The relative ease with which an attacker can modify the content or the appearance of a Web site, coupled with the ever-increasing penetration of Web-based applications into our daily life, demonstrates that there is an urgent need to develop methodologies for effectively addressing this issue.

In this article we discuss the design and experimental evaluation of a tool, which we call *Goldrake*, that may constitute the basis for a novel defacement detection service, in which a single organization could monitor the Web sites of hundreds or thousands of Web sites of other organizations. The crucial feature of Goldrake is that it does not require any participation from the monitored site. 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. The monitored site needs only provide a contact point that Goldrake will use for forwarding alerts, for example, a phone number and/or an email address. We remark that our proposal addresses only Web site defacements, that is, it is not meant to detect covert attacks aimed at spreading malware or stealing sensitive information [Provos et al. 2008]. Defense tools specialized to this purpose include [UCSB 2009; IBM Rational 2009; Dasient 2009; Ballard 2009]. As will be discussed in more detail, though, Goldrake does provide some monitoring ability on the components of a page not necessarily visible.

Our approach is based on *anomaly detection*. During a preliminary learning phase Goldrake automatically builds a profile of the monitored resource. Then, while monitoring, Goldrake will retrieve the remote resource periodically and generate an alert 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 [Ntoulas et al. 2004]. The challenge, thus, is to develop a tool capable of dealing with highly dynamic content while keeping

false positives to a minimum, while at the same time 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 not be fully complete or fully accurate.

Most Web sites lack a systematic surveillance of their integrity and the detection of Web defacements is often dependent on occasional checks by administrators or feedback from users. Indeed, a recent study on more than 60,000 defaced sites has found that 40% of the defacements lasted for more than one week and that the reaction time does not decrease significantly for sites hosted by Internet providers (and as such presumably associated with systematic administration) nor by taking into account the importance of sites as quantified by their PageRank [Bartoli et al. 2009]. 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 Section 6 for a more complete analysis). Essentially, all such technologies are based on a comparison between the 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. The main motivation for our work springs precisely from this observation: in our surroundings there are 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 [Kemp 2005]). 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. We remark that our proposal is not meant to replace the existing trusted copy approach. 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 and that this approach is intrinsically unable to

detect defacements provoked by attacks to the DNS infrastructure in which users are directed to fake sites [Wanjiku 2009; Danchev 2009; Mills 2009; DSL 2008]. Our proposal has instead the potential to detect those defacements (of course, provided users and our system share the DNS responses).

We built a prototype based on our approach and evaluated it on 300 dynamic resources observed every 6 hours for more than 4 months and on an attack set composed of 900 defacements extracted from a public archive of defacements. The results are very encouraging: Goldrake detected all the defacements (artificially) injected into the sequence of readings with a very low false positive rate—slightly less than one false alarm per week for each page, in a fully automatic setting. We also analyzed quantitatively and qualitatively performance and scalability of our proposal and we found that it appears indeed feasible.

2. OUR APPROACH

2.1 System Overview

A *Web resource*, resource for short, is a piece of data that is univocally identified by an URL—for example, an HTML document, an image file, and so on.

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, we 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*. 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) analyze r against its profile P_R;
(4) if r appears to be unusual, then execute some action.

The discussion of our prototype includes only the two most interesting parts: learning phase and analysis (step 3). The other steps of the monitoring phase can be understood easily. We point out, in particular, that the actual implementation of step 4 (how to send an alert to the monitored site and, at that site, how to handle the alert) is orthogonal to the topic of this article.

2.2 System Architecture

The resource R is analyzed by a number of *sensors*. A sensor characterizes one or more features of R, for example, its byte size or the number of links it contains. A sensor S can operate in either of two modes, as follows.

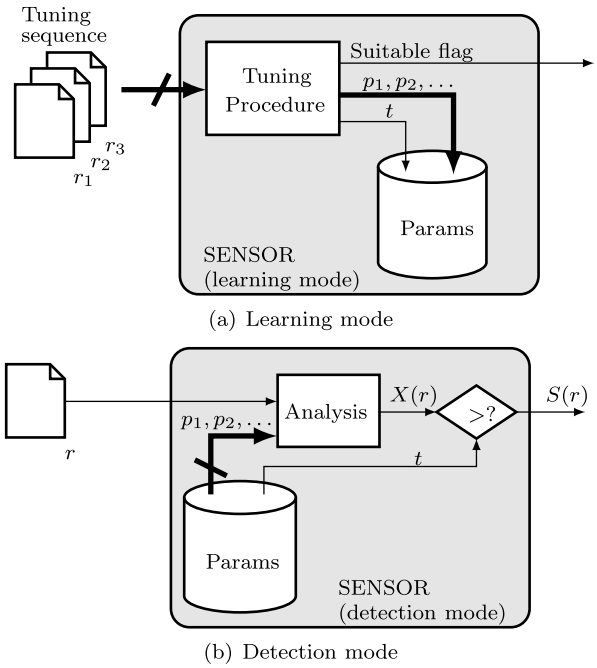

Fig. 1. Sensor internal architecture. Thin arrows indicate either numerical quantities or Boolean values. Thick arrows indicate data of other types, possibly not numerical.

In *learning mode*, S takes the tuning sequence and executes its own tuning procedure. As a result of this procedure, whose details are sensor-specific, S builds its own portion of the profile of R and verifies whether it is indeed able to operate with R. If not, S will state that it is *unsuitable* for R, in which case M will not apply S to R. This could be the case for a sensor that analyzes the relative frequencies of HTML elements in a resource that never contains any HTML element, for example, merely plain text, and for which it is hence not possible to compute relative frequencies of HTML elements.

In *detection mode*, S takes a reading r and returns a Boolean value $S(r)$ that tells whether r is consistent with the profile of R built in the learning phase. A true value means that S considers that r is not consistent with the profile. In this case we say that S raises an *alarm* or *alert*.

Internally, a sensor works as follows. The tuning procedure (Figure 1(a)) determines the values for a sensor-specific set of parameters $p_1, p_2, \ldots$ and for a numerical parameter t called *threshold*. In detection mode (Figure 1(b)), the current reading r is input to a parametric algorithm that produces a numerical index called *indicator* and denoted with $X(r)$. The parameters used for producing the indicator are those previously figured out during the learning phase, that is, $p_1, p_2, \ldots$. The value for $S(r)$ is then obtained by comparing $X(r)$ to the threshold: $S(r)$ is true if and only if $X(r) > t$.

The monitoring service M applies many different sensors on R. During the learning phase it attempts to use all the available sensors and then, in the

monitoring phase, it excludes only the sensors that declared themselves to be unsuitable for monitoring R.

In the monitoring phase the Boolean values produced by the sensors are combined together to produce a single Boolean, that will represent the overall categorization for the reading. We call this the *aggregator*, and denote it with A, the component of M which combines sensor outcomes. We denote by $A(r)$ the Boolean value output by A when the tool evaluates the reading r.

2.3 Our Prototype: Goldrake

We implemented a prototype, called Goldrake, according to the framework we have described. 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, 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 (see also Section 5).

2.3.1 *Sensors.* We implemented 45 sensors that can be grouped in 5 categories, as described below. In our prototype we focused on HTML documents, but some of our sensors can also work with other resources, such as RSS feeds. Below we denote by $\{r_1, r_2, \dots\}$ the tuning sequence of readings.

2.3.1.1 *Cardinality Sensors.* Each sensor in this category measures some simple numerical feature of the reading (e.g., the number of lines) and raises an alarm when the number is too different from what was expected.

The learning phase is as follows.

(1) Evaluate the measures $\{n_1, n_2, \dots\}$ on the readings of the tuning sequence.
(2) Compute mean η and standard deviation σ of the values obtained at the previous step.
(3) Set the threshold to $t = 3\sigma$.

A cardinality sensor never states it is unsuitable.

The indicator for a given reading r is computed as $X(r) = |n - \eta|$ where n is the measure for reading r.

This category includes 26 sensors, one for each of the features listed.

—Tags: block type (e.g., the measure performed by this 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, images whose names contain a digit, forms (all, external), tables, links (all, containing a digit, external, absolute).

For example, suppose that readings $\{r_1, r_2, r_3, r_4\}$ of the tuning sequence contain 75, 70, 83, and 77 links. The Cardinality sensor concerning links will compute $\eta = 76.25$ and $\sigma = 5.38$; then, the sensor will fire for a reading r containing n links if $n \leq 60$ or $n \geq 93$.

2.3.1.2 *FrequencyDifference Sensors.* Each sensor in this category computes the relative frequency of each item in a given item class (e.g., each character in the ASCII set, or each HTML tag in the set of all HTML tags) and raises an alert for a reading when the frequencies measured in that reading are too different from what expected.

We denote by $\mathcal{I}$ the item class of the sensor and by $n(I_i, r)$ the number of occurrences of item I_i in reading r. The learning phase is as follows.

(1) For each item $I_i \in \mathcal{I}$, count the number of occurrences $N(I_i)$ of that item in the tuning sequence $\{r_1, r_2, \ldots\}$: $N(I_i) = n(I_i, r_1) + n(I_i, r_2) + \ldots$
(2) For each item $I_i \in \mathcal{I}$, compute the relative frequency of item I_i in the tuning sequence:

$$f_i = \frac{N(I_i)}{\sum_{I_k \in \mathcal{I}} N(I_k)}. \tag{1}$$

(3) Compute the indicator $X(r)$ for each reading in the tuning sequence (the indicator is computed as explained below).
(4) Compute mean η_X and standard deviation σ_X of the indicators computed at the previous step.
(5) Set the threshold to $t = \eta_X + 3\sigma_X$.

A FrequencyDifference sensor states it is unsuitable for a given resource if and only if $f_i = 0, \forall i : I_i \in \mathcal{I}$, that is, when no items of the given class appear in any of the readings of the tuning sequence.

The indicator for a reading r is $X(r) = \sum_{i:I_i \in \mathcal{I}} |f_i^r - f_i|$ where

$$f_i^r = \frac{n(I_i, r)}{\sum_{I_k \in \mathcal{I}} n(I_k, r)} \tag{2}$$

are the relative frequencies of the items in the reading r being analyzed, computed as above.

This category includes two sensors. One analyzes characters contained in the visible textual part of the resource; the other analyzes HTML elements of the resource.

2.3.1.3 *MissingRecurrentItems Sensors.* We say that an item is recurrent when it appears in each reading of the tuning sequence. Each sensor in this category counts the number of missing recurrent items in a reading (details to follow) and raises an alert when that number is excessive.

The learning phase is as follows.

(1) Build the set R containing all items of the item class of the sensor that appear in each reading (i.e., $R = \{I : \forall r \in \{r_1, r_2 \ldots\}, I \text{ appears in } r\}$).
(2) Set the threshold $t = 0$.

A sensor in this category states it is unsuitable when $|R| = 0$, that is, when there are no items that can be found in each reading of the tuning sequence. Sensors of this category, thus, may not be suitable for some resources.

The indicator for a reading r is $X(r) = |R \setminus R_r|/|R|$ where R_r is the set containing all items of the item class associated with the sensor that appear in r. Since we set the threshold to 0, each sensor in this category assumes that even a single missing recurrent item is "too much." For example, if an embedded script appearing in each reading of the tuning sequence is not present in r, then the corresponding sensor will raise an alert. Note that, in this context, two scripts that differ in even one single statement are different scripts.

This category includes 10 sensors, each associated with one of the following item classes: images, images whose name contains one or more digits, embedded scripts, tags, words contained in the visible textual part of the resource and links. The link feature is considered as 5 different subfeatures, that is, by 5 different sensors of this group: all external, all absolute, all without digits, external without digits, absolute without digits.

For example, suppose that readings $\{r_1, r_2, r_3\}$ of the tuning sequence contain, respectively, external links $\{l_1, l_2, l_3, l_4\}$, $\{l_2, l_3, l_4, l_6, l_7\}$, and $\{l_1, l_2, l_4, l_6, l_8\}$. The MissingRecurrentItems sensor concerning external links will compute $R = \{l_2, l_4\}$; then, the sensor will fire for every reading r which does not contain at least external links l_2 and l_4.

2.3.1.4 *Tree Sensors.* Each sensor in this category builds a typical HTML/XML tree of the resource and raises an alert for a reading when the HTML/XML tree of that reading is excessively different from the typical one.

The learning phase is as follows.

(1) For each reading r_i in the tuning sequence $\{r_1, r_2 \dots\}$ build a tree $H(r_i)$ by applying a sensor-specific transformation on the HTML/XML tree of r_i (see also below).
(2) Build the tree $h = H(r_1) \cap H(r_2) \cap \dots$, where $a \cap b$ denotes the maximum common tree, containing the root node, between tree a and tree b.
(3) Set the threshold $t = 0$ (see the following).

A sensor in this category states it is unsuitable for a given resource if and only if $|h| < 2$, where $|h|$ is the number of nodes of the tree h.

The indicator for a reading r is $X(r) = 1 - |H(r) \cap h|/|h|$ where $H(r)$ is obtained applying the sensor-specific transformation on the HTML/XML tree of r. In other words, $X(r) = 0$ if $H(r)$ is a supertree of h; $X(r) = 1$ if $H(r)$ is not a supertree of any of the subtrees of h; for example, $|H(r)| = 0$ or $H(r)$ does not share any node with h. Since we set the threshold to 0, our prototype raises an alert for a reading whenever that reading is not a super-tree of h.

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 $H(r)$. Nodes of $H(r)$ contain only the type of the tag

(for example, TABLE could be a node of $H(r)$, whereas <TABLE CLASS="NAME"> could not).

—Only nodes of the HTML/XML tree of reading r that are tags in a predefined set (HTML, BODY, HEAD, DIV, TABLE, TR, TD, FORM, FRAME, INPUT, TEXTAREA, STYLE, SCRIPT) correspond to a node in the transformed tree $H(r)$. Nodes of $H(r)$ contain the full start tag (for example, <TD CLASS="NAME"> could be a node of $H(r)$, whereas <P ID="NEWS"> could not).

2.3.1.5 *Signature Sensors.* Each sensor in this category looks for a given attribute in a reading and raises an alert for that reading if that attribute is found. The indicator $X(r)$ for a given reading r is 1 when the attribute is found and 0 otherwise. The learning phase is as follows.

(1) Compute the indicator $X(r)$ for each reading in the tuning sequence.
(2) Set the threshold $t = 0$.

A sensor in this category states it is unsuitable for a given resource when $X(r_i) = 1$ for at least one reading r_i in the tuning sequence. Sensors in this category have no parameters depending on the resource being analyzed.

This category includes 5 sensors, one for each of the following attributes (rather common in defaced resources):

—has a black background;
—contains typical defacement phrases or known attackers' names;
—contains only one image or no images at all;
—does not contain any tags;
—does not contain any visible text.

Note that this category is radically different from the previous ones. While the former looks for signs of a defacement, the latter look for deviations from the profile of the resource. This difference resembles the distinction between the two complementary categories of intrusion detection systems: signature-based vs. anomaly-based.

2.3.2 *Aggregators.* We present here only the results that we obtained with one aggregator. Other examples can be found in Bartoli and Medvet [2006], Medvet et al. [2007], and Davanzo et al. [2008] (see also Section 5). It takes as input the outcomes of all sensors listed above (except for those that declared themselves as being unsuitable for the resource being monitored). The aggregator then counts the number n of categories for which at least one sensor fired and returns true if and only if $n \geq 4$, that is, if 4 or more categories have at least one sensor that fires.

We group the outputs of sensors basing on their category because sensors in the same category tend to exhibit similar results, that is, outputs of sensors in the same category tend to be strongly correlated. A legitimate, but substantial, modification in a resource may cause several sensors in the same category 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 Tree sensors nor MissingRecurrentItems sensors will fire.

3. EXPERIMENTS

3.1 Test Set

We performed all the experiments on an archive of 300 Web resources.[1] We accessed each resource every 6 hours for more than 4 months and recorded all the readings. Occasionally (0.056% of the number of readings) the HTTP responses were truncated or indicated that the server was unavailable. We fixed the corresponding reading by replicating the prior valid reading. The resulting archive has been used for simulating both the learning phase and the monitoring phase, which have been performed offline for convenience. Unless stated otherwise, the tuning sequence for a resource is composed of the first 50 readings for that resource.

The archive include technical Web sites (e.g., The Server Side, Java Top 25 Bugs), newspapers and news agencies both from the US and from Italy (e.g., CNN Business, CNN Home, La Repubblica, Il Corriere della Sera), e-commerce sites (e.g., Amazon Home), sites of the Italian public administration, the top 100 blogs from CNET, the top 50 Universities, and so on. Some resources were handpicked (mostly those in the technicals, e-commerce and USA newspapers groups) while the others were collected automatically by selecting the most popular resources from public online lists (e.g.: topuniversities.com for universities).

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 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.

3.2 Performance Indexes

We attempted to evaluate the ability of our tool to detect attacks, that is, major defacements. Unlike most works that evaluate IDSs [Lazarevic et al. 2003;

[1]The corresponding URL list is available at http://www.units.it/bartolia/download/Goldrake-archiveList.zip.

Lippmann et al. 2000; McHugh 2000], we do not have any standard attack trace to use. Therefore, we built a set of 900 different defacements, the *attack set*,[2] that we manually extracted from the Zone-H digital attack archive (http://www.zone-h.org). The set is composed of a selection of real defacements performed by different hackers or teams of hackers: we chose samples with different size, language, and layout and with or without images, scripts, and other rich features. We attempted to build a set with wide coverage and sufficiently representative of real-world defacements. In each experiment we submitted to Goldrake all the readings of the attack set and measured the False Negative Rate (FNR) as percentage ratio of number of missed detections to number of submitted attacks.

We also evaluated the ability of our tool to cope with legitimate dynamic content. To this end we simulated the monitoring phase by submitting to Goldrake the sequence of readings following the tuning sequence—without any defacement. We measured the False Positive Rate (FPR) as the percentage ratio of number of alerts to number of submitted readings, each alert raised in this scenario being a false alarm.

The FPR results are also cast in terms of False Positive Frequency (FPF), that is, the number of false alarms per unit of time (week). 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" (see also Section 3.8 and Section 4.2).

3.3 Performance without Retuning

In this section we analyse performance without retuning. That is, the tuning procedure is applied only once, at the end of the learning phase.

An important finding is that FNR is exactly zero. Even with a single tuning operation, Goldrake is able to detect all the defacements that we considered. On the other hand, FPR is very high and definitely unsatisfactory: 25.02% (averaged across all resources).

Interestingly, though, false alarms are not distributed uniformly over the monitoring period but are much more frequent near the end of the test. For example, FPR is approximately 3% during week 1, but grows to 14% and 34% during weeks 4 and 13 respectively. In fact, it appears that the profile built with the tuning procedure run at the end of the learning phase is sufficiently accurate, but only for a certain period. Eventually, the accumulation of changes makes that profile no longer useful.

For this reason, we enabled Goldrake to update the profile of a resource, through the execution of the tuning procedure also during the normal monitoring activity. The results are presented in the next section.

[2]Our selection is available online at http://www.units.it/bartolia/download/Goldrake-attackList.zip.

Table I. Average FPR, FPF, and FNR Data for Different Retuning Policies (Tuning Sequence Length of 50 Readings)

Policy	FPR %	FPF f.a./week	FNR %
FixedInterval ($n = 1$)	0.24	0.07	0.00
FixedInterval ($n = 15$)	1.67	0.47	0.00
FixedInterval ($n = 30$)	2.84	0.79	0.00
FixedInterval ($n = 45$)	4.19	1.17	0.00
UponFalsePositives	0.31	0.09	0.25

3.4 Evaluation of Retuning Policies

We implemented two *retuning policies*. Each policy specifies the conditions that trigger the execution of a tuning procedure. We used a tuning sequence length of 50 readings, like in the previous section. Goldrake maintains a sliding window including the 50 most recent readings. When the retuning procedure has to be applied, the tuning sequence is composed of all readings in that window.

—*FixedInterval*. The tuning procedure is applied every n readings. We experimented with $n = 1, 15, 30, 45$. This policy can be applied automatically, without any human intervention.

—*UponFalsePositives*. The tuning procedure is applied whenever Goldrake raises a false alarm. In other words, whenever Goldrake outputs a "true" 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 execute the tuning procedure before analyzing the next reading.

It is important to point out again that alerts are to be forwarded automatically to administrators of monitored sites. It follows that alerts in resources associated with the FixedInterval retuning policy do not require any handling by Goldrake operators. Concerning resources associated with the UponFalsePositive retuning policy, the retuning decision may be taken either by a Goldrake operator or by the administrator of the monitored site through a dedicated Web application. In the latter case, again, alerts do not require any handling by Goldrake operators. These considerations are essential to interpret the FPR results below and assess the scalability of our proposal (Section 4).

Table I summarizes the results, averaged across all resources. Goldrake still exhibits FNR=0% when using the *FixedInterval* policy, that is, it retains the ability of detecting every defacement included in the attack set. FNR is still very good but slightly worse with human-triggered retuning.

The key result is that FPR becomes tolerable thanks to retuning. The tuning procedure involving a human operator achieves FPR=0.31%, as opposed to 25.02% without retuning. Even fully automated procedures perform very well: a retuning performed at every reading leads to FPR=0.24%, whereas retuning every 15 readings (about 4 days) leads to 1.64% FPR.

The results for the three better retuning policies are shown separately for each resource in Figure 2. It can be seen that with FixedInterval ($n = 15$) there

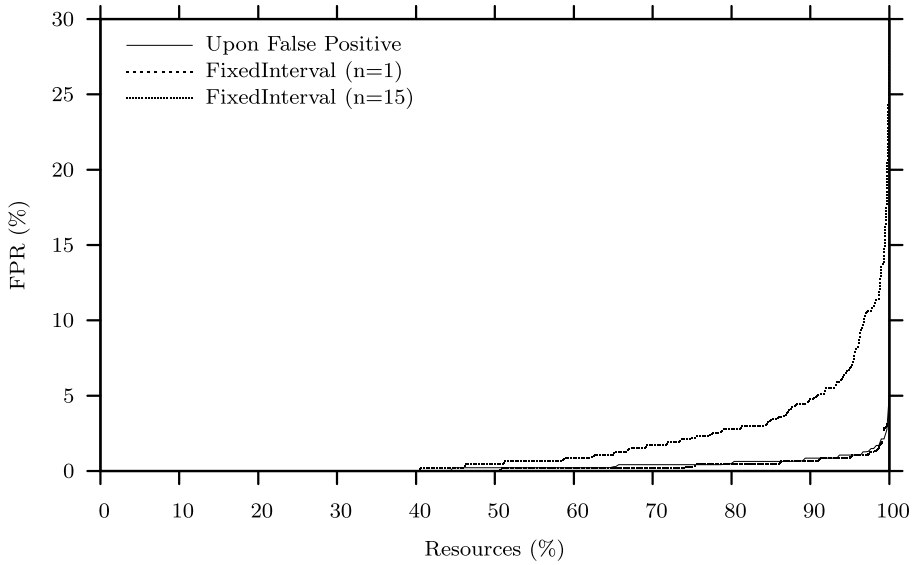

Fig. 2. Percentage of resources with FPR smaller than the given FPR value.

is a non-negligible fraction of resources for which FPR is not satisfactory. With the two other retuning policies, however, there are no intractable resources.

3.5 Impact of Tuning Sequence Length

The results of the previous section have been obtained with a tuning sequence composed of 50 readings. We repeated the previous experiments with other tuning sequence lengths in the range [5, 100]. The detection ability turned out to be almost independent of the tuning sequence length: FNR results were essentially identical to those described in Section 3.4, except for an increase to 0.05% with human triggered retuning with a very short tuning sequence ($l \leq 10$ readings). FPR results with FixedInterval ($n = 1$) and UponFalsePositive are shown in Figure 3. It can be seen that Goldrake is quite robust with respect to variations in the tuning sequence length, as a few days of preliminary observations suffice to build an adequate profile. Interestingly, 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 5).

3.6 "Fast Start" Working Mode

Increasing the tuning sequence length may reduce FPR, as demonstrated in the previous section. This option, however, has the drawback of increasing the time required for bootstrapping the tool: the learning phase will last longer, thus the actual monitoring will start later. For example, in our setting, with a

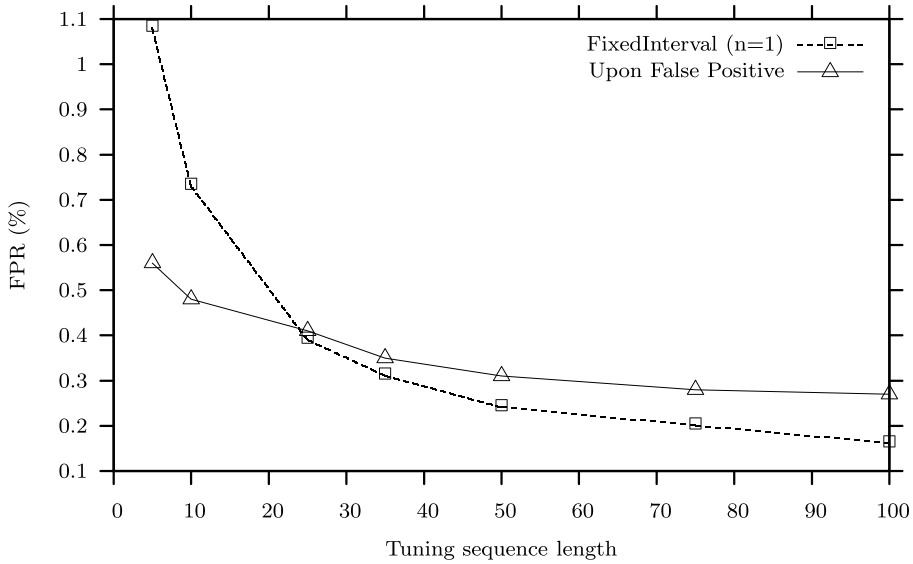

Fig. 3. Average FPR with different tuning sequence length on two policies.

tuning sequence length $l = 90$ readings the learning phase will last for 23 days, whereas with $l = 50$ readings the learning phase will last for 13 days.

In this section we describe and evaluate a working mode, which we call *fast start*, aimed at shortening the bootstrap time. In this working mode the monitoring phase starts after 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.

In fast start Goldrake executes the first tuning procedure on a sequence of l_i readings and then starts monitoring. A retuning is performed every 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, that is, with $l = l_f$.

We experimented with FixedInterval ($n = 1$), with fast start parameters $\Delta l = 1$, $l_i = 5$, and $l_f = 50$. That is, we started with a tuning sequence length of 5 readings, thereby starting monitoring after only 30 hours; we incremented the tuning sequence length by 1 at each reading ($\Delta l = 1$) until reaching the target length of 50 readings. With this setting, the fast start phase lasted for almost 13 days, that is, one increment every 6 hours for 45 times.

As a comparison we ran FixedInterval ($n = 1$) in normal mode, that is, without fast start, with the two extreme values for the tuning sequence length: first with $l = l_i = 5$ readings, then with $l = l_f = 50$ readings. In the former case the tool begins the monitoring phase with the same (very short) initial delay, but

Table II. Effect of the Fast Start Working Mode. Mean FPR When Using the FixedInterval Policy with Different Values for the Tuning Sequence Length, $n = 1$

	average FPR (%)	
reading #	1...50	51...100
FastStart ($l = 5 \ldots 50$)	0.47	0.26
FixedInterval ($n = 5$) ($l = 5$)	1.19	0.99
FixedInterval ($n = 50$) ($l = 50$)	n.a.	0.26

Table III. Raw Sensor Data, Grouped by Categories

	Unable			By Sensor		By Category	
Category	average	max.	std.dev.	FPR	FNR	FPR	FNR
Tree (2)	0.00	0	0.00	0.47	0.30	0.65	0.27
FrequencyDifference (2)	0.00	0	0.00	6.38	0.08	10.50	0.00
Signature (5)	0.02	1	0.13	0.00	61.42	0.01	3.08
Cardinality (26)	0.00	0	0.00	2.75	10.39	21.19	0.00
MissingRecurrentItems (10)	0.52	8	1.05	5.53	5.86	28.39	0.00

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 (almost 13 days). To highlight the effects introduced by fast start, we evaluate FPR at the very beginning of the monitoring period. The results are in Table II, where FPR is computed on the first 50 readings submitted after the tuning sequence and then on the following 50 readings (after those readings the effects introduced by fast start vanish).

It can be seen that the fast start mode is very effective. For the same initial delay, it provides a remarkable FPR reduction—for example, 0.47% vs. 1.19%. Once it has reached the final tuning sequence length, it provides the same FPR as the normal mode, that is, 0.26%.

3.7 Effectiveness of Sensor Categories

Useful insights about our approach can be obtained by analyzing the behavior of individual sensors, that is, before the merging operated by the aggregator. For ease of discussion, we average values across all resources and group the results by category.

Table III ("Unable" column) shows the number of sensors which stated to be unsuitable for the given resource after the initial tuning procedure. It can be seen that Tree, FrequencyDifference and Cardinality sensors could be applied to each resource. There were some resources for which one Signature sensor could not be applied and some resources for which up to 8 MissingRecurrentItems could not be applied. Looking at the raw data in detail, it turns out that the full set of sensors was applied on 248 resources.

Table III ("By Sensor" column) summarizes the FPR and FNR that one would obtain with each sensor taken in isolation, with retuning policy FixedInterval ($n = 1$). The values in the By Category column are instead those that one would obtain with each category taken in isolation. These results suggest several important observations. First, there is no single category providing good

performance on both FPR and FNR. Each category provides reasonable performance only on one of the two indexes and Tree sensors appear to provide the best trade-off in this respect. Second, the behavior of all categories but Tree is sensibly different at the sensor level and at the category level. In particular, Cardinality and MissingRecurrentItems have excellent detection ability at the category level (FNR=0) but not at the sensor level. On the other hand, their detection ability as categories is obtained at the expense of a significant FPR increase. Finally, we note that Signature sensors are the least effective ones in detecting attacks. In other words, at least in our experimental setting, looking for signs of a defacement is less effective than detecting deviations from a profile. For the defacement detection problem, thus, 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). This somewhat surprising conclusion, however, could be biased by the small number of sensors in our Signature category and could deserve further analysis.

3.8 Higher-Frequency Monitoring

In all our tests we used a monitoring interval of 6 hours. With this setting, thus, Goldrake would detect a defacement within 6 hours after its occurring. A prompter detection may be obtained by simply decreasing the monitoring interval. The lower bound on the monitoring interval ultimately depends on the available resources (see also Section 4.2).

It is important to point out that decreasing the monitoring interval does not necessarily result in a corresponding increase of the false positive frequency FPF. For example, a resource exhibiting FPR=1% when checked every 6 hours would result in approximately FPF=0.28 f.a./week. Checking that resource every hour would not necessarily increase FPF by six times, though. The reason is because FPR itself is likely to depend on the sampling frequency. Indeed, collecting more readings in a given tuning interval may lead to a resource profile of better quality which may in turn improve FPR—an effect observed in the experiments below. Of course, this consideration cannot be taken to its extreme consequences: sampling a resource many times a second would hardly improve the quality of the training data due to the resulting strong correlation among training samples.

To assess the overall effect on FPF, we performed a suite of experiments with a very short monitoring interval: 5 minutes, corresponding to 288 readings every day for each resource. We performed these experiments on 13 resources.[3] In this experiment we chose to focus on a small number of resources for practical reasons. We attempted to fetch a much larger number of resources, but we found that in many cases we were banned by the Web server, most likely because of our particular access pattern—short and nearly constant time interval between two requests, same source IP. Indeed, this pattern was classified as abnormal also by our academic network policy.

[3]The list of these resources is available online at http://www.units.it/bartolia/download/Goldrake-higherFrequency.zip. This set is nearly identical to the set of resources that we monitored in our earlier work [Bartoli and Medvet 2006].

We executed retuning at every reading (FixedInterval with $n = 1$) and used the same tuning sequence length as in the previous experiments ($l = 50$). This configuration corresponds to an excessively short time window for constructing the profile: $5 \cdot 50 = 250$ minutes rather than $6 \cdot 50 = 300$ hours. For this reason, we experimented with another decimated configuration in which we constructed the tuning sequence by taking one new reading every 6 hours. In other words, we experimented with two configurations: (i) a tuning sequence composed of a sliding window of the 50 most recent readings, as usual; and (ii) a tuning sequence constructed by including a new reading every 6 hours. In the decimated configuration, thus, the time window used for constructing the profile has the same length as in the previous experiments.

The results show that the approach is practical even in this nearly extreme case. The baseline values for the resources in this experiment, computed with the previous 6 hours monitoring interval, are FPR=0.37% and FPF=0.10 false alarms/week/resource. The results obtained with 5 minutes monitoring interval are FPR=0.04% in normal configuration and 0.01% in decimated configuration, corresponding to FPF=0.79 and 0.20 false alarms/week/resource respectively. It can be seen that FPF remains very small although the monitoring frequency increases 60 times. Most importantly, the increase in FPF may be kept low by simply decoupling the monitoring interval from the the time interval for constructing the profile—in the decimated configuration we may greatly lower the former while keeping the latter constant.

3.9 Detection of Minor Modifications

All the experiments performed so far are based on a broad sample of real-world defacements. As it turns out, defacements tend to be quite different from genuine Web pages. A question that naturally arises is whether our approach may also detect also subtler defacements constrained on smaller portions of the page. In this Section we attempt to answer this question by focussing on the ability of Goldrake to detect subtler unauthorized modifications of a page, which we generated systematically by *merging* portions of real-world defacements with the page.

For a given resource, we constructed a profile based on a tuning sequence of 50 readings. We then generated a number of randomly modified readings from the next genuine reading r, as follows. We replaced randomly chosen subtrees of r by randomly chosen subtrees of a defacement. We accumulated several such changes until a specified percentage w of r was changed. We measured this percentage, which we call *merging rate*, on the byte size of the resource. We generated 50 randomly modified readings from r, 10 for each of the 5 merging rates used: 10%, 20%, 30%, 40%, 50%—for example, a merging rate $w = 40\%$ means that 40% of the code of the resource has been modified. We allowed for a tolerance of $\pm 5\%$ in merging rates, because matching exactly a given merging rate value was often not possible. We executed this procedure with 110 different defacements, submitted each of the modified readings to Goldrake, and counted each missed detection as a false negative. We repeated the experiment on 24 resources, evaluating approximately 132,000 modified readings. Resources and

Table IV. Missed Detections of Minor Modifications (FNR) as a Function of the Merging Rate. The Three Columns Refer to Goldrake, Human Subjects, and Goldrake When Analyzing Only Readings Detected by at Least One Human Subject

Merging Rate %	Goldrake %	Humans %	Goldrake on humans %
10	51.97	24.24	52.11
20	23.39	11.16	26.78
30	11.39	2.98	12.42
40	7.27	0.45	8.96
50	5.68	1.79	5.98

defacements were selected at random from our archive. The results are given in Table IV ("Goldrake" column). It seems reasonable to claim that Goldrake may be a very effective detection tool even when the defacement retains a substantial part of the original page content. For example, even when 70% of the page remains unchanged, Goldrake detects 89% of the modified readings.

As expected, FNR increases when the merging rate decreases. Assessing which is an "acceptable" FNR threshold is difficult, however, because not all modified readings could satisfy the specific goal of an attacker. Moreover when the merging rate is very low the visual modification impact becomes lighter and, consequently, the notion of defacement itself becomes fuzzier. For example, inserting a pseudonym somewhere in a page as a proof of successful intrusion could be a hacker's goal, yet such a change would cause considerably less damage than a major disruption of the page and could arguably not qualify as a defacement.

In the attempt of providing a baseline for these results we performed a second suite of experiments involving human subjects, as follows. In each experiment we selected a resource R at random and generated two sequences: a sequence S_1 with 5 normal readings of R and a sequence S_2 with a randomly shuffled mix of 25 readings of R. S_2 consisted of 10 normal readings and 15 modified readings. All readings were randomly selected from those in the given dataset. We carefully verified that all the modified readings involved the visible portion of the page, such that the subject were potentially able to detect it. The subject was shown S_1 first and then S_2. The subject was instructed that S_1 contained normal readings of a Web resource and was asked to tell, for each reading in S_2, whether the reading was normal or had been modified by a hacker. We involved subjects of varying age and skill by disseminating requests along several channels—our mail contacts, Facebook contacts, students, and so on. We allowed each subject to perform multiple experiments, at his will, and were able to collect 112 experiments from 97 users, totaling 1676 evaluations for modified readings—we considered only the experiments which provided at least 24 evaluations.

The results are given in Table IV ("Humans" column). Not surprisingly, human subjects perform better than Goldrake, especially at small merging rates. The key result, though, is that detection of modified readings at small merging rates is a challenging task even for human subjects—135 and 50 missed detections over 557 and 450 readings with $w = 10\%$ and $w = 20\%$, respectively. This

result is significant also because of the very same nature of the experiment: subjects knew they had to look for unauthorized changes, which made them particularly sensitive. Of course, these results have to be interpreted with care because they do not consider active evasion attempts by attackers. This topic is analyzed in Section 5.

Finally, we performed a third suite of experiments by evaluating the detection rate of Goldrake limited to the 1373 modified readings that were detected by at least one subject—modified readings not detected by any subject can hardly qualify as defacements. The results ("Goldrake on humans" column) are essentially identical to those obtained in the first suite of experiments, with the full set of modified readings. This result indicates that the detection ability of Goldrake is not skewed toward pages that are hardly perceived as defacements.

4. 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.

4.1 Number of Resources

In this section we attempt to estimate the maximum number of resources that can be monitored, say N_{MAX}. Recall that alerts are to be forwarded automatically to administrators of monitored sites. The value of N_{MAX} depends mainly on the retuning policy used, as follows.

Concerning resources associated with a fully automatic retuning policy, alerts do not require any handling by Goldrake operators. It seems reasonable to claim, thus, that N_{MAX} is limited only by the hardware resources available. Addition of a new monitored resource may be done by the administrator of that (remote) resource, with the help of a dedicated Web application, for example. 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 retuning decision may be taken either by a Goldrake operator or by the administrator of the monitored site through a dedicated Web application. The scalability bottleneck, thus, is given by false alarms in those resources whose retuning decision is demanded to Goldrake operators.

Let k denote the number of operators at Goldrake involved in the handling of those resources. Let A_{FPF} denote the False Positive Frequency exhibited by Goldrake for a single resource and let T_{FPF} denote the maximum False Positive

Table V. Maximum Number of Monitored Resources N_{MAX} for Each Operator, for Several Retuning Frequencies T_{FPF} (Resources Whose Retuning is in Charge of Goldrake Operators)

T_{FPF} f.a./day	T_{FPF} f.a./week	N_{MAX}
1	7	78
2	14	156
4	28	311
20	140	1556

Frequency that can be tolerated by a single human operator. Assuming that all resources exhibit the same A_{FPF}, we obtain the following estimate for N_{MAX}:

$$N_{MAX} \cong k\frac{T_{FPF}}{A_{FPF}}. \tag{3}$$

According to our findings in the previous sections, we can safely assume for A_{FPF} a value of 0.09 false positives per week, on each resource; (in fact, our experiments exhibit 0.09 f.a./week with UponFalsePositive retuning, see Table I). Finding a suitable value for T_{FPF}, on the other hand, is clearly more difficult and hard to generalize. Table V 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 only 4 false positives per day—probably an excessively low estimate—then he may monitor 311 resources. From a different point of view, if one monitors 311 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 or delegated to administrators of monitored sites; capable of monitoring several hundreds of resources per operator, for resources whose retuning is human-triggered and delegated to Goldrake operators. Obviously, an actual deployment of the service would have several options available regarding the retuning policies offered. Configurations that require the involvement of Goldrake operators could even be discarded.

4.2 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 (i) there is no overlap at all among downloading, evaluation and retuning of different resources; and (ii) a retuning is performed at every reading for each of the N monitored resources. It will be $m_l > (t_d + t_e + t_r) \cdot N$.

Table VI. Minimum Monitoring Interval m_l (mins:secs) for N Resources Monitored on Our Test Platform

N	No overlap	Full overlap
78	2:42	0:02
156	3:24	0:03
311	7:48	0:04
1556	34:58	0:17

The average values for t_d, t_e, and t_r that we found in our experiments are, respectively, 1300msecs, 100μsecs, and 10msecs. It follows that 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 is shown in Table VI (no overlap column). For example, if Goldrake monitors 311 resources—without any overlap among download and analysis of different resources—then an attack may be detected in approximately 8 minutes. From a different point of view, if one requires a detection time in the order of 8 minutes, then no more than 311 resources should be monitored.

In practice there may be a substantial overlap between download and analysis—Goldrake may download a resource while analyzing other resources, for example. A less conservative estimate may thus be constructed by assuming some degree of overlap. In the extreme case of full overlap between download and analysis, we would obtain $m_l > (t_r+t_e)\cdot N$. Since the download time t_d is predominant over $t_r + t_e$, the estimated lower bound for m_l becomes much smaller as shown in Table VI (full overlap column).

We remark that: (i) these figures refer to a single computing node; monitoring of different resources are independent jobs, hence the monitoring task is linearly scalable in the number of computing nodes; and (ii) any bounds due to involvement of Goldrake operators in human-triggered retuning policies are completely orthogonal to this analysis.

5. DISCUSSION

5.1 Detection of Attacks and Large-Scale Deployment

Experiments clearly show the excellent detection abilities of Goldrake. With automatic retuning FNR is exactly zero; that is, Goldrake detected *all* defacements in our attack set. 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" changes to a resource, such as those corresponding to today's defacements. The experiments in Section 3.9, moreover, show that Goldrake may be effective even when the change is focussed on limited portions of a resource.

It is however important to assess potential strengths and weaknesses of our proposal if it were deployed on a large scale and became a sort of nearly ubiquitous defacement detection mechanism. We assume that: (i) site administrators will react promptly to alerts; and (ii) the duration of a defacement is a key component of the attackers' goal.

It seems reasonable to claim that our system would force attackers to rethink their attack strategies radically. Today, attackers may deface a site by inserting fraudulent content at their will, in particular, with the freedom of selecting that content in a way fully independent of the legitimate content of the attacked site. It seems likely that this kind of attack will no longer be effective, based on the breadth of our attack set and our excellent results in terms of false negatives. It follows that attackers will no longer have complete freedom in choosing the fraudulent content and, most importantly, will be forced to perform a preliminary analysis of the site under attack in order to (try to) craft a defacement that will not be flagged as anomalous. We believe this basic fact is a valuable contribution of our proposal.

Our system also appears to constitute a powerful approach for demoting the current generation of automatic defacement tools.[4] These tools have enabled a radically novel and potentially disruptive attack pattern: rather than trying to deface a specific Web site, locate thousands of Web sites that exhibit a certain vulnerability and thus can certainly be defaced. Once these sites have been located, they can all be defaced simultaneously with a few keystrokes.[5] These tools are likely to become ineffective, as they are based on the pattern previously mentioned: apply the same defacement to sets of sites that suffer from the same vulnerability. It seems very unlikely that any given defacement, constructed without any prior analysis of the specific site to be attacked, will fit the profile of many different sites. The outcome of our experiments [Medvet et al. 2007] indirectly corroborates this claim further: once the system has been trained for a given resource, the system strongly tends to flag as anomalous any reading of other resources.

In summary, our proposal may constitute the first significative step forward taken by defenders in the defacement arena, that so far has been under full control of attackers. A successful deployment of our proposal on a large scale, on the other hand, is likely to cause attackers to evolve their strategies. The next issue to discuss, thus, is assessing whether our proposal may play this adversarial game effectively.

We make three preliminary observations. First, the set of sensors used in this article is essentially the result of our early two-weeks conceptual design of the system. We did not care to implement more sensors because performance in our benchmarks was good enough already. A production version of the system could and should include many more sensors than those presented here—for example, more sensors focussed on Javascript/Ajax content, on HTML tables, `IFRAMES` and so on. An area where Goldrake could be improved is the analysis of textual content, as none of our sensors attempts to build a profile of

[4]"A Commercial Web Site Defacement Tool," Dancho Danchev's Blog, Mind Streams of Information Security Knowledge, April 2008 (http://ddanchev.blogspot.com); "MultiInjector v0.3 released," Chapters in Web Security Blog, November 2008 (http://chaptersinWebsecurity.blogspot.com).

[5]"Know your Enemy: Web Application Threats" The Honeynet Project and Research Alliance, February 2007 (http://www.honeynet.org/papers/Webapp/); "Hacking Methodologies - An overview of historical hacking approaches," (Johnny Long, http://www.scribd.com/doc/191122/Hacking-Methodologies).

common sentences or of the language used, for example. Obviously, aggregators will have to accommodate the new sensors so as to maintain FPR low, but we have found out that this is not particularly difficult to achieve with a careful aggregator design [Medvet et al. 2007; Davanzo et al. 2008] (see also the following). It follows that Goldrake is not a *static* defense that, as such, would become nearly useless if attackers learnt how to circumvent it. On the contrary, defenders may extend and improve their detection coverage according to their skills, experience, and emerging menaces. This ability has proven crucial in large-scale adversarial problems of practical interest, for example, spam filtering [Goodman et al. 2007; Cormack and Lynam 2007; Ramachandran et al. 2007].

Second, one may define many different aggregators with very similar benchmark performance but that enforce significantly different notions of anomalous reading. Let A_D denote the specific aggregator analyzed in this article. Let A'_D denote an aggregator constructed by OR-ing the output by A_D and the output of the RecurrentScripts sensor—a MissingRecurrentItems sensor which outputs true for reading r when any of the embedded scripts included in every reading of the tuning sequence is missing from r (Section 2.2). We have implemented this variant, and it turns out that the two aggregators, A_D and A'_D, exhibit essentially identical FPR/FNR values in all the operational conditions analyzed so far (we omit the details for brevity). The resulting variants of Goldrake, though, enforce quite different notions of anomalous reading, as with A'_D any change in a recurrent embedded script will certainly raise an alert.

The same technique may be used with any sensor whose output exhibits a sufficiently low FPR. In our experiments there are further 8 sensors which, taken in isolation, have both FPR and standard deviation (across the 300 resources) smaller than 1%. The monitoring of resource R could thus start with A_D and then one could tailor the actual $A'_D(R)$ dynamically, based on the FPR exhibited by individual sensors on R—essentially, the firing of sensors that rarely fire on R may be made a sufficient condition for generating an alert on R. The notion of anomalous reading may be changed also based on the threshold values for sensors. For example, we have implemented another variant in which the output of A_D is OR-ed with the output of instances of Tree sensors in which the threshold is set to 1 rather than to 0 (as it is in A_D, Section 2.3.1). Once again, we obtained a benchmark performance equivalent to that of A_D.

We have also synthetized aggregators by means of machine learning techniques that generate automatically a function of sensors' output so as to minimize FPR + FNR—we used *genetic programming* [Koza 1992] in Medvet et al. [2007] and *support vector machines* [Boser et al. 1992] in Davanzo et al. [2008]. These aggregators exhibit benchmark performance equivalent to that of A_D, but their structure is such that they enforce sensibly different notions of anomalous readings. Further examples are given in Section 5.3.

As a third and final preliminary observation, existing literature and practical experience about security-related adversarial games demonstrate that keeping parts of the system secret may greatly strengthen the defenses. In fact, this approach is often both necessary and feasible. In intrusion detection

systems it is usually believed that full knowledge of the detection algorithm allows developing techniques for circumventing detection [Handley et al. 2001; Tan et al. 2003]. In detection systems based on forms of machine learning—like ours—the standard security practice consists of assuming that the learning algorithm is common knowledge whereas the set of features used by a specific instance of the algorithm is kept secret, which is feasible in systems with a small number of deployments [Barreno et al. 2006]. A materialization of this principle can be found in spam filtering: while the general principles of spam filtering tools are well known, the exact details and settings of the tools used, say, by Gmail or Hotmail, are kept secret. Indeed, making these details public would only benefit attackers.[6] It is thus reasonable to assume that, in a large-scale deployment of our proposal, part of the detection machinery has to be kept secret.

Attackers could attempt to construct and run locally a copy of the system. This approach does not seem to help very much, because there are simply too many (secret) sensors and aggregators that could be used by the system. Guessing all such components, even leaving aside their settings, seems to be hardly feasible.

Attackers could try to gain knowledge about the system details by probing, that is, hacking into, monitored sites. In principle, the attacker could reverse engineer all the system details with a sufficiently large and carefully selected set of probes. We are not able to quantify the amount of work that would be necessary to this purpose. We argue, however, that such an approach would hardly be practical, due to the size and complexity of the design space resulting by all possible sensors and aggregators. The design space magnifies further as the system may be configured to use different features for different resources. Tailoring the configuration to each specific resource, for example at the aggregator level, may be useful for attempting to "tighten" the resource profile. This could be done automatically, already discussed, or based on directives provided by administrators of the monitored resource—for example, the presence of an obscene word anywhere in a reading could be made a sufficient condition for generating an alert (see also Section 5.3). Indeed, the defense strategy could purposefully include randomization elements in the system configuration [Barreno et al. 2006].

Leaving full reverse engineering aside, attackers could still try to gain sufficient knowledge about a specific target site by means of probing. This is a relatively common activity for spammers and requires the execution of an "optimization loop" on the system they wish to circumvent [Graham 2003; Goodman et al. 2007]. In our scenario defacers would have to cope with a loop where each single iteration lasts many minutes and whose outcome is hard to interpret: absence of reaction to an intrusion could be due to an undetected intrusion, but also to an administrator not reacting promptly to alerts, as well as to a longer-than-usual monitoring cycle. When the probing activity provokes detected intrusions, moreover, absence of reaction to further probes could be due

[6]This specific point is addressed in more depth and compared with design principles of cryptographic systems in "Secrecy, Security and Obscurity," B. Schneier, *Crypto-Gram*, 2002.

to disinformation purposefully operated by administrators of the site for confusing the attacker [Barreno et al. 2006]. It seems reasonable to claim that the attackers' task would be far from easy. Furthermore, and most importantly, this laborious task does not lend itself well to being automated and much of it would have to be repeated for each new target.

Of course, one cannot take for granted that attackers will never manage to slip through our detection system. The key observation in this respect is that, as pointed out already, our approach is not a static defense and the detection coverage may be improved as a reaction to novel menaces. This improvement could even be performed systematically. An aggregator tailored to the detection of a specific family of defacements, to be OR-ed with the other aggregators, could be synthesized automatically with a traditional learning algorithm—for example, support vector machine, or neural network, or genetic programming. Any technique that may be trained on positive samples is a candidate, as we have done [Medvet and Bartoli 2007; Davanzo et al. 2008]. The generation of a new aggregator of this kind could be triggered by a human operator or even automatically, for example as a reaction to a mass defacement undetected by our system but somehow notified by a sufficient amount of monitored sites—a pattern very close to the "report spam" feature of GMail (see also Ramachandran et al. [2007] and Goodman et al. [2007]). Although we have not implemented this functionality yet, we believe it is a very promising direction.

Another issue to consider in a large-scale setting is the quality of the learning data. A key requirement for any anomaly-based intrusion detection system is that the data used for building the resource profile are indeed attack-free. In practice the absence of attacks in the learning set is usually taken for granted or verified "manually." While such a pragmatic approach may be feasible in a carefully controlled environment, it is clearly infeasible in a large-scale scenario like ours. For this reason, we have proposed a procedure for automatic detection of corrupted data in a learning set that works very effectively even with corruption rates up to 50% [Medvet and Bartoli 2007]. We artificially inject defacements into a learning set whose quality is unknown and evaluate the classification performance before and after the injection. As it turns out from our experimental evaluation, any strong change in classification performance is an excellent indicator of the presence of corruption in the learning set. We developed this procedure for coping with the starting phase of the monitoring process, but the procedure may be executed online periodically and even at each retuning.

The quality of the learning data could also be used as an attack tool. As with any other anomaly-based intrusion detection system, the attacker might alter the learning data so as to progressively shift the region classified as normal toward a region of his choice [Barreno et al. 2006]. We believe a very promising defense against this form of attack may be constructed based on the procedure just mentioned. All executions of the procedure may refer to the same test sequence, hence the evolution of classification performance across successive procedure executions may provide strong indications about the nature of the learning set. When the difference before and after corruption becomes less sensible, then a specific alert could be raised. The procedure could be executed

also by injecting into the learning set a static set of readings collected at service subscription time. In this case the artificial corruption should not introduce significant alterations in classification performance.

5.2 False Positives

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, our key contribution consists of the retuning policies. Even with a fully automatic retuning performed every $n = 30$ readings (approximately once a week), FPR remains moderate: 2.79%, which corresponds to approximately 3 false alarms per month per resource. By retuning more frequently FPR improves further: 1.64% with $n = 15$ and 0.24% when retuning at every reading ($n = 1$). Interestingly, these figures are essentially equivalent to those of human-triggered retuning. Indeed, fully automatic retuning is advantageous also from the FNR point of view (Table I). 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 also be very useful if we could somehow quantify 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 as 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: 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 [Mutz et al. 2006; Michael and Ghosh 2002; Shavlik and Shavlik 2004; Ye et al. 2002; Lazarevic et al. 2003]. 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 [Mutz et al. 2006].

It is interesting to observe that we obtained very low FPR despite the simplicity of our algorithms. We tune resource-specific parameters in a very simple

way, for example, the threshold is either a constant value or a predefined function of mean and standard deviation of the indicator. We rebuild profiles either at regular intervals or when a false alarm occurs. We have experimented with more sophisticated anomaly-detection techniques that have been applied to Intrusion Detection Systems [Ramaswamy et al. 2000; Breunig et al. 2000; Mahalanobis 1936; Ye et al. 2002; Yeung and Chow 2002; Mukkamala et al. 2002; Lazarevic et al. 2003], but they do not appear to improve our results significantly: the corresponding performance range from roughly equivalent (with support vector machines) to worse or much worse, depending on the specific technique [Davanzo et al. 2008].

Finally, we note that our structuring of sensors and aggregator makes it easy to generate intuitive explanations of alerts automatically. This property is very important in practice, in particular for dealing with false positives. Providing easy to interpret reports with other anomaly-detection approaches—for example, based on support vector machines or neural networks—is much harder and often not possible.

5.3 Personalization

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 firing of such a sensor could also be made a sufficient condition for raising an alert, irrespective of the output of the other sensors and irrespective of the actual content of the tuning sequence. 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. Some U.S. Government Web sites were successfully attacked in this way [McMillan 2007].

Similar requirements may emerge from many other application domains. 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. Or the system could be configured so as to raise an alert upon any change in the Javascript content. Such forms of personalization could be part of different levels of service offered by the organization running Goldrake. One could even monitor a resource with several different aggregators, which could be useful for attaching a form of severity level to each alert depending on the aggregator that raised that alert. Note that all the above examples refer to portions of the page content that are not necessarily visible.

6. RELATED WORK

We first describe existing tools specifically devoted to Web defacement detection (Section 6.1) and then other tools that could be used for the same goal (Section 6.2). Next we survey works that analyze similarities and differences in Web content over time (Section 6.3). We discuss the relation between our

approach and intrusion detection (Section 6.4), anomaly detection (Section 6.5) and, finally, spam detection (Section 6.6).

6.1 Tools for Web Defacement Detection

Several tools for automatic detection of Web defacements exist and some of them are or have been commercially available. They are generally meant to be run *within* the site to be monitored (e.g., WebAgain,[7] Sitegrity[8]). Some of these tools may be run remotely (e.g., Catbird[9]), 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 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 [Sedaghat et al. 2002; Fone and Gregory 2002]. 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—for example, 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. It seems thus practically unavoidable that any trusted copy approach should consider only *parts* of the resource, which 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 and maintain the related technology; and (ii) integrate the technology with the daily Web-related operational process of the organization—for example, streaming the up-to-date resource content to the trusted copy. As we 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

[7]http://www.lockstep.com
[8]http://www.breach.com
[9]http://www.catbird.com

organizations simply lack an automated and systematic surveillance of the integrity of their Web resources. Our approach attempts to address precisely such organizations. A surveillance service that can be joined with just a few mouse clicks and that have minimal impact on the daily Web-related operations would certainly have the potential to achieve a vast diffusion. We 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.

6.2 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 [Kim and Spafford 1994]. 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 [Pennington et al. 2003; Banikazemi et al. 2005; Sivathanu et al. 2005; Gehani et al. 2006].

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.

6.3 Web Similarity and Change Detection

Our tool analyzes a Web resource based on its content and appearance. Similar analyses have been proposed for different purposes. SiteWatcher [Liu et al. 2006; Fu et al. 2006] 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 Goldrake we are concerned also with attacks directed at resource features that might not affect the visible look of the resource. WebVigil [Sanka et al. 2006] is a tool closer to ours 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 from our work 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 our work we first build a profile of the resource automatically, then we compare the resource, as currently available, against its profile.

Important insights about the temporal evolution of Web resources have been provided in Fetterly et al. [2004]. The cited work presents a large-scale study, involving 151 millions Web resources observed once a week for 11 weeks. They used a similarity measure between resources based on the notion of shingle [Broder et al. 1997], a sequence of words excluding HTML markups (see the cited papers for details). Such a measure is quite different from ours; 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 a significant fraction of Web resources do not change over time. When changes do occur, moreover, these usually affect only the HTML markups and do so in minor ways (addition/deletion of a tag and the like). 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 dynamicity of a resource, at least in the short-term, by observing the resource for some period. From our point of view, this property is important for "tightening" the profile appropriately.

These results are confirmed by Ntoulas et al. [2004], who 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 nonnegligible fraction of resources. These observations confirm our (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.

6.4 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 [Mutz et al. 2006; Chari and Cheng 2003], network traffic in Network-based IDSs (e.g., Snort http://www.snort.org, [Chang et al. 2001; Heberlein et al. 1990]), application messages or events in Application Protocol-based IDSs [Kruegel and Vigna 2003; Anitha and Vaidehi 2006]. In our case we analyze instead the *state* of the monitored system. Unlike storage-based IDS and storage integrity checkers (see Section 6.2)

that access the internal state, moreover, we analyze 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 detect intrusions only 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. Our findings 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), and 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 [Michael and Ghosh 2002; Sekar et al. 2001]. The notion of "state" in such approaches is quite different from ours. We reason on the externally observable state of the system, rather than on the state of an automaton constructed in a training phase. From this point of view our approach is more similar to some emerging techniques for invariant-based bug detection. The technique described in Chilimbi and Ganapathy [2006] consists of inspecting periodically the content of the heap during program execution—that is, 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).

6.5 Anomaly Detection

Our work borrows many ideas from *anomaly-based* IDSs [Denning 1987; Gosh et al. 1998; Kruegel et al. 2002; Zanero and Savaresi 2004]. Broadly speaking, we observe a system to learn its behavior and raise an alert whenever we detect something unusual. Clearly, the technical details are very different: rather than observing network traffic or system call invocations, we observe Web resources.

A framework able to detect anomalous system call invocations is proposed in Mutz et al. [2006]. 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 focused 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 our setting; hence, we had to face to problem of finding an algorithm for deciding when and how to update the profile of a resource.

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 Kruegel and Vigna [2003] 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. As in the previous case, profile updating is not treated.

Two comparative studies of several anomaly detection techniques for intrusion detection are provided by Lazarevic et al. [2003] and Patcha and Park [2007], while techniques based on statistical classification are presented in Shyu et al. [2003] and Ye et al. [2002]. 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. As observed in Section 5.3, our approach compares favorably to such techniques [Davanzo et al. 2008].

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 Shavlik and Shavlik [2004], which describes a system for host intrusion detection in which the profile of the user is retuned periodically. In our scenario this form of retuning has turned out to be a necessity and thus constitutes an essential component of our approach.

6.6 Spam Filtering

Our approach to Web defacement detection exhibits interesting similarities to *spam detection*, that is, the problem of separating unsolicited bulk electronic messages from legitimate ones [Cranor and LaMacchia 1998]. In both cases, one wants to classify an item as being "regular" or "anomalous" without defining in detail what 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 [Cormack and Lynam 2007]. Such tools exploit signatures for the content of spam messages and may have to update such signatures to follow new trends in spammers' behaviors [Goodman et al. 2007]. Approaches based on machine learning have also been proposed [Androutsopoulos et al. 2000].

Broadly speaking, spam filtering differs from Web defacement detection in the cost of false positives and negatives. 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—for example, 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—for example, PageRank [Page et al. 1998]. A framework for detecting spam posts in blog resources is proposed in [Mishne et al. 2005]: 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 our approach is in the nature of the profile: they focus on a single, sophisticated, feature based on the textual content of the resource, whereas we consider a number of simple and easy to construct features.

7. CONCLUSIONS

We have discussed the design and experimental evaluation of a novel service for the detection of Web defacements, a very common form of attack. The service may monitor thousands of Web sites of remote organizations systematically and automatically, with minimal operator support and horizontal scalability. As such, it is especially amenable to be hosted in a cloud facility. An organization may join the service with just a few mouse clicks, without installing any software locally or changing its daily operational processes. The service does not need any prior knowledge about the monitored sites, yet it allows exploiting such knowledge when available.

These features may greatly facilitate a widespread diffusion of our proposal, even at small organizations, thereby modifying today's very common scenario where the detection of Web defacements is not a systematic activity but depends on occasional checks by administrators or feedback from users. Our proposal constitutes the only alternative to existing technology, providing a radically different trade-off in terms of operational costs and detection capability. Moreover, our approach is more robust towards defacements provoked by attacks to the DNS infrastructure. The two approaches, though, complement each other and could even constitute multiple layers of a defense-in-depth strategy.

Our service is based on anomaly detection. We have assessed its performance on a sample composed of 300 dynamic resources observed every 6 hours

for more than 4 months and on an attack set composed of 900 defacements extracted from a public archive, with very good results in terms of FNR and FPR. These results appear to demonstrate the practical feasibility of the approach, which is also able to generate easy to interpret reports automatically.

We have elaborated on possible consequences of a deployment of our system on a large scale. We have shown that our proposal could force attackers to rethink their strategies radically, which would constitute the first significative step forward taken by defenders in the defacement arena. We have also focussed on likely evolutions of attack strategies and devised several evolutions of our system that may allow playing the resulting adversarial game effectively.

ACKNOWLEDGMENTS

The authors are grateful to Fulvio Sbroiavacca, who provided the initial motivations for this work, and to the anonymous reviewers for their constructive comments.

REFERENCES

ANDROUTSOPOULOS, I., KOUTSIAS, J., CHANDRINOS, K. V., AND SPYROPOULOS, C. D. 2000. An experimental comparison of naive Bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'00)*. ACM Press, New York, 160–167.

ANITHA, A. AND VAIDEHI, V. 2006. Context based application level intrusion detection system. In *Proceedings of the International Conference on Networking and Services (ICNS'06)*. IEEE Computer Society, Los Alamitos, CA, 16.

BALLARD, L. 2009. Show me the malware! Google online security blog. http://googleonlinesecurity.blogspot.com/2009/10/show-me-malware.html.

BANIKAZEMI, M., POFF, D., AND ABALI, B. 2005. Storage-based file system integrity checker. In *Proceedings of the ACM Workshop on Storage Security and Survivability (StorageSS'05)*. ACM Press, New York, 57–63.

BARRENO, M., NELSON, B., SEARS, R., JOSEPH, A. D., AND TYGAR, J. D. 2006. Can machine learning be secure? In *Proceedings of the ACM Symposium on Information, Computer, and Communications Security*. ACM, 16–25.

BARTOLI, A. AND MEDVET, E. 2006. Automatic integrity checks for remote Web resources. *IEEE Intern. Comput. 10*, 6, 56–62.

BARTOLI, A., MEDVET, E., AND DAVANZO, G. 2009. The reaction time to Web site defacements. *IEEE Intern. Comput. 13*, 4, 52–58.

BOSER, B. E., GUYON, I. M., AND VAPNIK, V. N. 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory*. 144–152.

BREUNIG, M. M., KRIEGEL, H.-P., NG, R. T., AND SANDER, J. 2000. LOF: Identifying density-based local outliers. *SIGMOD Rec. 29*, 93–104.

BRODER, A. Z., GLASSMAN, S. C., MANASSE, M. S., AND ZWEIG, G. 1997. Syntactic clustering of the Web. *Comput. Netw. ISDN Syst. 29*, 8-13, 1157–1166.

CERT/CC. 2001. FedCIRC Advisory FA-2001-19 "Code Red" worm exploiting buffer overflow in IIS indexing service DLL. Advisory, US-Cert. http://www.us-cert.gov/federal/archive/advisories/FA-2001-19.html.

CHANG, H.-Y., WU, S. F., AND JOU, Y. F. 2001. Real-time protocol analysis for detecting link-state routing protocol attacks. *ACM Trans. Inform. Syst. Secur. 4*, 1, 1–36.

CHARI, S. N. AND CHENG, P.-C. 2003. BlueBoX: A policy-driven, host-based intrusion detection system. *ACM Trans. Inform. Syst. Secur. 6*, 2, 173–200.

CHILIMBI, T. M. AND GANAPATHY, V. 2006. HeapMD: Identifying heap-based bugs using anomaly detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. ACM Press, New York, 219–228.

CORMACK, G. V. AND LYNAM, T. R. 2007. Online supervised spam filter evaluation. *ACM Trans. Inform. Syst. 25*, 3.

CRANOR, L. F. AND LAMACCHIA, B. A. 1998. Spam! *Comm. ACM 41*, 8, 74–83.

DANCHEV, D. 2009. Hackers hijack DNS records of high profile New Zealand sites. *ZDNet*. http://blogs.zdnet.com/security/?p=3185.

DASEY, D. Oct. 2007. Cyber threat to personal details. *The Sydney Morning Herald*. http://www.smh.com.au/news/technology/cyber-threat-to-personal-details/2007/10/13/1191696235979.html.

DASIENT. 2009. Dasient Web anti-malware. http://www.dasient.com/.

DAVANZO, G., MEDVET, E., AND BARTOLI, A. 2008. A comparative study of anomaly detection techniques in Web site defacement detection. In *Proceedings of the 23rd International Information Security Conference*. 711–716.

DENNING, D. E. 1987. An intrusion-detection model. *IEEE Trans. Softw. Engin. 13*, 2, 222–232.

DSL. 2008. Comcast domain hacked. *DSLReports.com*. http://www.dslreports.com/shownews/Comcast-Hacked-94826.

FETTERLY, D., MANASSE, M., NAJORK, M., AND WIENER, J. L. 2004. A large-scale study of the evolution of Web pages. *Softw. Pract. Exper. 34*, 2, 213–237.

FONE, W. AND GREGORY, P. 2002. Web page defacement countermeasures. In *Proceedings of the 3rd International Symposium on Communication Systems Networks and Digital Signal Processing*. IEE/IEEE/BCS, 26–29.

FU, A. Y., WENYIN, L., AND DENG, X. 2006. Detecting phishing Web pages with visual similarity assessment based on earth mover's distance (EMD). *IEEE Trans. Depend. Secur. Comput. 3*, 4, 301–311.

GEHANI, A., CHANDRA, S., AND KEDEM, G. 2006. Augmenting storage with an intrusion response primitive to ensure the security of critical data. In *Proceedings of the ACM Symposium on Information, Computer, and Communications Security (ASIACCS'06)*. ACM Press, New York, 114–124.

GOODMAN, J., CORMACK, G. V., AND HECKERMAN, D. 2007. Spam and the ongoing battle for the inbox. *Comm. ACM 50*, 2, 24–33.

GORDON, L. A., LOEB, M. P., LUCYSHYN, W., AND RICHARDSON, R. 2006. 2006 CSI/FBI Computer Crime and Security Survey. Security survey, Computer Security Institute.

GOSH, A. K., WANKEN, J., AND CHARRON, F. 1998. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 14th Annual Computer Security Applications Conference (ACSAC'98)*. IEEE Computer Society, Los Alamitos, CA, 259.

GRAHAM, P. 2003. Better Bayesian filtering. http://www.paulgraham.com/better.html.

HANDLEY, M., PAXSON, V., AND KREIBICH, C. 2001. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th Conference on USENIX Security Symposium*. USENIX Association.

HEBERLEIN, L. T., DIAS, G. V., LEVITT, K. N., MUKHERJEE, B., WOOD, J., AND WOLBER, D. 1990. A network security monitor. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, 296.

IBM RATIONAL. 2009. *Malware Scanner Extension for IBM Rational AppScan*. http://www.ibm.com/developerworks/rational/downloads/08/appscan_malwarescanner/index.html.

KEMP, T. 2005. Security's Shaky State. *Inform. Week*. http://www.informationweek.com/industries/showArticle.jhtml?articleID=174900279.

KIM, G. H. AND SPAFFORD, E. H. 1994. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS'94)*. ACM Press, New York, 18–29.

KIRK, J. 2007. Microsoft's U.K. Web site hit by SQL injection attack. *ComputerWorld Security*. http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9025941.

KOZA, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press.

KRUEGEL, C., TOTH, T., AND KIRDA, E. 2002. Service specific anomaly detection for network intrusion detection. In *Proceedings of the ACM Symposium on Applied Computing (SAC'02)*. ACM Press, New York, 201–208.

KRUEGEL, C. AND VIGNA, G. 2003. Anomaly detection of Web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM Press, New York, 251–261.

LAZAREVIC, A., ERTÖZ, L., KUMAR, V., OZGUR, A., AND SRIVASTAVA, J. 2003. A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of the 3rd SIAM International Conference on Data Mining*. SIAM, San Francisco, CA.

LIPPMANN, R., HAINES, J. W., FRIED, D. J., KORBA, J., AND DAS, K. 2000. Analysis and results of the 1999 DARPA offline intrusion detection evaluation. In *Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection (RAID'00)*. Springer-Verlag, 162–182.

LIU, W., DENG, X., HUANG, G., AND FU, A. Y. 2006. An antiphishing strategy based on visual similarity assessment. *IEEE Intern. Comput. 10*, 2, 58–65.

MAHALANOBIS, P. C. 1936. On the generalized distance in statistics. In *Proceedings of the National Institute of Science of India*, 12, 49–55.

MCHUGH, J. 2000. Testing intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Trans. Inform. Syst. Secur. 3*, 4, 262–294.

MCMILLAN, R. 2007. Bad things lurking on government sites. *InfoWorld*. http://www.infoworld.com/article/07/10/04/Bad-things-lurking-on-government-sites_1.html.

MEDVET, E. AND BARTOLI, A. 2007. On the effects of learning set corruption in anomaly-based detection of Web defacements. In *Proceedings of the 4th GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA)*. Springer.

MEDVET, E., FILLON, C., AND BARTOLI, A. 2007. Detection of Web defacements by means of genetic programming. In *Proceedings of the 3rd International Symposium on Information Assurance and Security*. IAS, Manchester, UK.

MICHAEL, C. C. AND GHOSH, A. 2002. Simple, state-based approaches to program-based anomaly detection. *ACM Trans. Inform. Syst. Secur. 5*, 3, 203–237.

MILLS, E. 2009. Puerto Rico sites redirected in a DNS attack. *CNET*. http://news.cnet.com/8301-1009_3-10228436-83.html.

MISHNE, G., CARMEL, D., AND LEMPEL, R. 2005. Blocking blog spam with language model disagreement. In *Proceedings of the 1st International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*.

MUKKAMALA, S., JANOSKI, G., AND SUNG, A. 2002. Intrusion detection using neural networks and support vector machines. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'02)*. 1702–1707.

MUTZ, D., VALEUR, F., VIGNA, G., AND KRUEGEL, C. 2006. Anomalous system call detection. *ACM Trans. Inform. Syst. Secur. 9*, 1, 61–93.

NTOULAS, A., CHO, J., AND OLSTON, C. 2004. 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*. ACM Press, New York, 1–12.

PAGE, L., BRIN, S., RAJEEV, M., AND TERRY, W. 1998. The PageRank Citation Ranking: Bringing Order to the Web. Tech. rep., Stanford University.

PATCHA, A. AND PARK, J.-M. 2007. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw. 51*, 12, 3448–3470.

PENNINGTON, A. G., STRUNK, J. D., GRIFFIN, J. L., SOULES, C. A., GOODSON, G. R., AND GANGER, G. R. 2003. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*. USENIX.

PREFECT. 2010. Congressional Web site defacements follow the state of the union. *Praetorian Prefect*. http://praetorianprefect.com/archives/2010/01/congressional-Web-site-defacements-follow-the-state-of-the-union/.

PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. 2008. All your iframes point to us. In *Proceedings of the 17th Conference on Security Symposium (SS'08)*. USENIX Association, 1–15.

PULLIAM, D. Aug. 2006. Hackers deface federal executive board Web sites. http://www.govexec.com/story_page.cfm?articleid=34812.

RAMACHANDRAN, A., FEAMSTER, N., AND VEMPALA, S. 2007. Filtering spam with behavioral blacklisting. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. ACM, New York, 342–351.

RAMASWAMY, S., RASTOGI, R., AND SHIM, K. 2000. Efficient algorithms for mining outliers from large data sets. *SIGMOD Rec. 29*, 427–438.

RICHARDSON, R. 2007. 2007 CSI Computer Crime and Security Survey. Security survey, Computer Security Institute.

SANKA, A., CHAMAKURA, S., AND CHAKRAVARTHY, S. 2006. A dataflow approach to efficient change detection of HTML/XML documents in WebVigiL. *Comput. Netw. 50*, 10, 1547–1563.

SEDAGHAT, S., PIEPRZYK, J., AND VOSSOUGH, E. 2002. On-the-fly Web content integrity check boosts users' confidence. *Comm. ACM 45*, 11, 33–37.

SEKAR, R., BENDRE, M., DHURJATI, D., AND BOLLINENI, P. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'01)*. Los Alamitos, CA, 144.

SHAVLIK, J. AND SHAVLIK, M. 2004. Selection, combination, and evaluation of effective software sensors for detecting abnormal computer usage. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*. ACM Press, New York, 276–285.

SHYU, M.-L., CHEN, S.-C., SARINNAPAKORN, K., AND CHANG, L. 2003. 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 3rd IEEE International Conference on Data Mining (ICDM'03)*. IEEE, 172–179.

SIVATHANU, G., WRIGHT, C. P., AND ZADOK, E. 2005. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the ACM Workshop on Storage Security and Survivability (StorageSS'05)*. ACM Press, New York, 26–36.

SMITH, G. Feb. 2007. CRO Website hacked. *Silicon Republic*. http://www.siliconrepublic.com/news/news.nv?storyid=single7819.

TAN, K., MCHUGH, J., AND KILLOURHY, K. 2003. Hiding intrusions: From the abnormal to the normal and beyond. In *Revised Papers from the 5th International Workshop on Information Hiding*, Lecture Notes in Computer Science, vol. 2578. 1–17.

UCSB. 2009. Wepawet—on line Web malware detection. http://wepawet.cs.ucsb.edu.

WANJIKU, R. 2009. Google blames DNS insecurity for Web site defacements. *Infoworld*. http://www.infoworld.com/t/authentication-and-authorization/google-blames-dns-insecurity-Web-site-defacements-722.

YE, N., EMRAN, S. M., CHEN, Q., AND VILBERT, S. 2002. Multivariate statistical analysis of audit trails for host-based intrusion detection. *IEEE Trans. Comput. 51*, 7, 810–820.

YEUNG, D.-Y. AND CHOW, C. 2002. Parzen-window network intrusion detectors. In *Proceedings of the 16th International Conference on Pattern Recognition*. 385–388.

ZANERO, S. AND SAVARESI, S. M. 2004. Unsupervised learning techniques for an intrusion detection system. In *Proceedings of the ACM Symposium on Applied Computing (SAC'04)*. ACM Press, New York, 412–419.

ZONE-H. 2006. Statistics on Web Server Attacks for 2005. http://www.zone-h.org.

Received October 2008; revised July 2009, November 2009, March 2010; accepted March 2010