*Article*

# Defacement Detection with Passive Adversaries

**Francesco Bergadano [1,*], Fabio Carretto [2], Fabio Cogno [2] and Dario Ragno [2]**

[1] Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy
[2] Certimeter Group, Corso Svizzera 185, 10149 Torino, Italy
* Correspondence: francesco.bergadano@di.unito.it; Tel.: +39-011-6706743

Received: 28 May 2019; Accepted: 25 July 2019; Published: 29 July 2019

**Abstract:** A novel approach to defacement detection is proposed in this paper, addressing explicitly the possible presence of a passive adversary. Defacement detection is an important security measure for Web Sites and Applications, aimed at avoiding unwanted modifications that would result in significant reputational damage. As in many other anomaly detection contexts, the algorithm used to identify possible defacements is obtained via an Adversarial Machine Learning process. We consider an exploratory setting, where the adversary can observe the detector's alarm-generating behaviour, with the purpose of devising and injecting defacements that will pass undetected. It is then necessary to make to learning process unpredictable, so that the adversary will be unable to replicate it and predict the classifier's behaviour. We achieve this goal by introducing a secret key—a key that our adversary does not know. The key will influence the learning process in a number of different ways, that are precisely defined in this paper. This includes the subset of examples and features that are actually used, the time of learning and testing, as well as the learning algorithm's hyper-parameters. This learning methodology is successfully applied in this context, by using the system with both real and artificially modified Web sites. A year-long experimentation is also described, referred to the monitoring of the new Web Site of a major manufacturing company.

**Keywords:** adversarial learning; anomaly detection; defacement response; Security Incident and Event Management; Security Operations Center

---

## 1. Introduction

Web Site and application defacement is considered a major security incident, that should be detected as early as possible. It requires appropriate response technology and processes, and a number of approaches have been proposed in the scientific literature [1–11]. Commercial services have also been proposed (e.g., those surveyed in [12], but see also Weborion and PharmingShield), as well as software products [e.g., Nagios, Sawmill, Foresight, NGINX], that can be adapted to this purpose.

The consequences of undetected defacements can be far-reaching for any organization's image and operations. Yet, appropriate detection and response is often lacking, as is demonstrated by average reaction times [13], and by the large number and importance of reported incidents [14,15]. The Zone-H monitoring portal [16] maintains a listing of reported Web Site defacements, together with relevant statistics. During the past eight years, an average of one million defacements per year have been reported [15].

Defacement detection can be addressed as an anomaly detection problem: defaced Web content can be labeled as anomalous, based on a classifier that is learned from previously available examples [1,2,5,17]. However, this is not normally sufficient, because an adversary may observe the alarm-raising behaviour of the anomaly detector, and try to find wanted defacements that will escape detection. It is therefore necessary to prevent an adversary from knowing what will be classified as anomalous and what will not. Although some works have addressed this issue (see, e.g., [6,18]), it still

represents a limitation of the state-of-the-art solutions and of the above-cited previous research in defacement detection.

This paper proposes a general way of dealing with passive adversaries in defacement detection, by developing a generalized concept and methodology of keyed learning, that can be defined as learning with a key that is unknown to an adversary. The key will influence the learning process in many possible ways—for example, it will determine the choice of the learning algorithm and its parameters. A detailed description of the key's effect on the defacement detection system will be provided in Section 3. A consequence of this approach is that the adversary will not have the information necessary to mimic the learning process, using available training data, and obtain a learned hypothesis that is similar to the one we will use.

One way to prevent the prediction of a learned classifier by an adversary is to hide the hypothesis space, hyper-parameters, or the learning bias [6], or by introducing some form of randomization [18–20]. We argue in the present paper that many other factors that influence learning could be hidden from our adversary, including the subset of the available training data that are really used, and the timing and context of the learning process. In order to avoid so-called "security through obscurity", we concentrate the information hidden from our adversary in a secret key, whilst the overall learning procedure is left open and public. In this paper we define a precise and general way to derive all such hidden parameters from a secret key, and to apply them in a comprehensive anomaly detection system.

The concept of keyed intrusion detection has been introduced in the literature [21–25], with similar motivations. This was, however, restricted to that particular application domain, and the secret key was used only as a "word delimiter" in intrusion payloads. Keyed learning was first highlighted in a previous conference presentation [26], and is a more general concept than keyed intrusion detection, because it affects any form of learning and uses any kind of secret information. In the present paper we develop this idea with further detail (Section 3).

We have implemented a keyed and adaptive defacement detection system, and successfully deployed it and tested it in a full-size and real-world context. The results show that the approach is simple and feasible, and leads to a reasonable number of alerts. The alerts that were generated are unpredictable, if the key is not known, but at the same time they were found to be well motivated and really corresponding to anomalous content. We take a practical approach: our methodology is based on application requirements and on previous experience in learning classifiers and anomaly detection [26,27]—we will then start by describing the general context of defacement response.

## 2. Defacement Response

Defacement response is a complex process, that is planned based on risk analysis results and available budget. It normally comprises three phases:

- *Prevention*. One should in the first place avoid the very possibility of defacement, by protecting the relevant perimeter, including the Content Management Systems (CMS), and the target Web Servers and Proxies. This can be done with conventional security measures, e.g., WAFs, IDS/IPS systems, vulnerability scanning and remediation.
- *Detection*. When defacement occurs it should be immediately detected, yielding low error rates for both false positives (e.g., normal CMS changes, acceptable external feeds, and advertising banners), and undetected defacements.
- *Reaction*. Again, a three-phase process:

    1. Alerting: based on detection alarm thresholds, an alert is generated, firing automated reaction or a request for human intervention.
    2. Inspection: defacements are rare, yet high priority events; as a consequence, human inspection is needed when an above-threshold alarm is fired. Integration with a Security Operation Center (SOC) and corresponding Security Incident and Event Management (SIEM) software architectures is needed for enterprise-level solutions. The reaction component

should be available as a 24/7 service and appropriate information/escalation procedures should be defined.

3. Mitigation: a courtesy page should be displayed, until normal operation is enforced. If proxies or a CDN such as Akamai are involved, this action should be propagated accordingly.

In this paper, we will concentrate on the detection phase. See [28] for a discussion of the general relationship between *detection* and *response* in anomaly detection. We will, however, also address issues that may be relevant for the reaction phase.

### 2.1. Defacement Detection: Problem Definition

From a practical point of view, defacement detection should raise an alarm and activate the reaction phase when a Web Site or an Application interface is "not what it is supposed to be". As such, it is an ill-defined problem, as there is no way to formalize a definition of what the page is supposed to look like.

One could define accepted content based on a previously accepted reference page, or on some application image that has been separately stored (see, e.g., [10], and most of the commercial monitoring services [12]). However, the web site's pages will change often, and modification processes are normally outsourced to suppliers and system integrators, involving some complex CMS and corresponding authorizations to third parties. Hence, detection based on a separately maintained reference content, with an equality check, is practically impossible to implement and deploy in an enterprise context. At the same time, this very system could be attacked and made ineffective [14]: a hacker could take control of both the Web Site and of the CMS, or the client machines where users insert content changes could be compromised. Finally, active proxies could be targeted.

We would then like to define a possibly defaced application as a system producing information that does not respond to one or both of the following criteria:

- Matching a set of predetermined rules, e.g.,:
  - including/excluding a set of keywords/images
  - responding to semantic consistency [12]
  - responding to a given graphical/content schema
- Response to some similarity criterion:
  - it should be similar to a set of previously published or otherwise accepted content, and/or
  - it should be different from known unwanted content, or from content samples that have been labeled as out of context.

Whether this possibly defaced application actually is a defacement incident can only be assessed by human intervention. This could be labeled as a form of Turing test, and one could argue that no totally automated system can detect defacements without errors. In a relatively recent defacement incident, an international airport Web Site was changed, and departures were substituted with unlikely tropical destinations - indeed a difficult one for an automated defacement detection system. As a consequence, a reasonable detection/reaction approach for enterprise solutions can be heuristic and based on previously accepted content:

- a current page version is downloaded using undeclared IP sources, hidden client types and randomized navigation patterns, so as to avoid so-called *differential defacement* [14], where an adversary tries to detect monitoring before defacement is enacted;
- a matching degree w.r.t. the above-mentioned predetermined rules is computed;
- a similarity value w.r.t. accepted and, possibly, unwanted pages is computed;
- an overall heuristic "defacement degree" is obtained;

- if this number is above a given threshold, a SOC service is alerted, with possible human intervention. The threshold should be set based on available SOC budget, and on risk level considerations (e.g., in the context of ISO-27001 or other standards and regulations). The best automated detection system will yield low error rates: a small percentage of "false positives" (reducing SOC effort), and a very high percentage of alerts being fired when a real defacement has occurred (reducing risk).

The ill-defined defacement detection problem can now be understood:

**Defacement:** the unauthorized change of a Web Site or a Web Application front-end, that introduces significant modifications, with important negative impacts for the reputation and the operations of the owner organization

**Defacement detection:** the act of promptly realizing a defacement has occurred, followed by the generation of an alarm and a severity score

A definition of what a "significant" modification should be depends on risk management choices, often based on business impact and regulatory context. For example, changing the company logo or name, or inserting totally extraneous text, is normally considered a serious defacement, while the normal upgrade of Web site content and of the graphical evolution of the interface, as well as the changing advertising banners, should not generate continuous alarms.

The above definitions must be understood in the context of a defacement response process, where prevention measures make the occurrence of a defacement instance unlikely, and alarm management and mitigation strategies are in place, normally with the support of an SOC. In this context, we are interested in two error measures for the defacement detector during a defined period of time:

**UDR (Undetected Defacement Rate):** the relative number of defacement occurrences, validated as such by human SOC operators, but not recognized by the defacement detector

**FAR (False Alarm Rate)** : the relative number of generated alarms that do not correspond to real defacements

We may then define the following requirements for a defacement detector:

**Very low UDR.** Undetected defacements have extreme consequences, so their number should be as low as possible.

**Low FAR.** False alarms cost, as they must be dealt with by the SOC. Based on available budget, thresholds can be set so as to have a desired FAR value. Obviously, lower FARs will make the expected UDR higher.

**Unpredictability.** Only system administrators, responsible for the defacement detector, should have access to details and keys that make defacement alarms predictable. If adversaries know the alarm behaviour, they will be able to simulate it and break the defacement response infrastructure.

### *2.2. Previous Work in Defacement Detection*

A number of papers strictly related to defacement detection and response have been published [1–11]. Some commercial monitoring services are also available [12].

The simplest approach is based on direct comparison of downloaded content with trusted and separately stored content [4,8,10,12]. Most commercial services [12] detect any minor change and fire an email alert, based, for example, on the difference in the hash codes of downloaded and trusted content. The same approach is used in [10], where the check is embedded in a ready-to-use Apache module. The module will prevent a defaced page from being served to a client. A similar method, based on a more involved difference computation, is described in [8]. The authors have done some experimentation and have realized that such difference checks can be computationally demanding, hence they propose efficient but partial difference evaluation algorithms. In [4] an extreme approach is adopted, where reference content is stored in a read-only local storage, so as to avoid the very

possibility of content injection that could be so pervasive as to avoid detection. Similarly, in [11] the operating system's features are used, in combination with a secure hardware component, to monitor file systems changes and detected unwanted modifications.

More involved approaches, that are more directly related to the present paper, are adaptive in nature, and rely on Pattern Recognition and Machine Learning techniques, in order to classify a currently downloaded page as either "normal" or "defaced" (e.g., [1,3,5–7]). As such, these approaches are part of a more general notion of anomaly detection, with applications to computer security. We will not cite papers from this broader area, but excellent recent surveys are available, e.g., [28,29]. In [6] a number of different or parametric approaches to anomaly detection for defacement monitoring are surveyed and quantitatively compared. In particular, the use of different learning algorithms is evaluated in the light of performance (false positive and false negative rate).

In [1] a novel "Delta Framework" is described, using a Machine Learning approach, and introducing a concept of "Fuzzy tree difference". This is interesting, because it overcomes the limitations of an exact match, and it addresses the issue of the computational workload that such monitoring services typically involve. In both [1] and [9] linguistic and semantic issues are addressed. In particular, in [9] the notion of "semantic consistency" is introduced, a notion that can be effective for detecting undercover defacements, where page look and feel is maintained, but nonsense, or totally out of context sentences are injected.

Quite in the opposite direction, the approach described in [2] is based on image difference and can be effective when text is unchanged or not involved in the defacement attack, because everything is obtained with the final visual effect facing the user.

Defacement approaches based on image recognition are challenged with a very difficult goal, being related to a kind of Turing test, this time a form of "reversed CAPTCHA":

- in a standard CAPTCHA, the "good guy" is a human user, who reads and types text from an image. The "bad guy" is an automated algorithm, also trying to correctly *read* some words, in order to bypass robot detection.
- in this reversed CAPTCHA, applied to image-based defacement detection, the "good guy" is an automated algorithm (the monitoring system), trying to extract meaning from an image and checking that it is consistent with the site's profile. The "bad guy" is a human attacker, wanting to *write* unwanted text in the image.

Since this reverse CAPTCHA challenge is intrinsically difficult, most defacement detection approaches are not based on image recognition and labeling.

## 3. Keyed Learning of a Defacement Detector

Many security challenges have been addressed by means of adaptive techniques, involving both supervised learning and clustering. These include one step and especially continuous authentication, network intrusion detection, spam filtering, defacement response and malware analysis (see, e.g., [29,30]). In such applications examples are often continuously generated, and online learning methods are needed.

In anomaly detection challenges, a security adversary may avoid defense mechanisms by influencing or predicting the output of every learning phase. Following the seminal paper by Kearns and Li [31], and more recent work [18,20–25,32–35], a new field of research has emerged, known as adversarial learning. As a large number of examples, both real and malicious, may be generated, the term "adversarial data mining" has also been used [36].

Defacement response has not been directly studied in an adversarial context, except for the inspiring discussion in the final section of [6]. However, it should be: our adversary is willing to deface our application, and will maximize success by trying to pass undetected through our defacement detection component. This can involve two lines of action: (1) influencing the learning system, e.g., by injecting malicious examples, so that it will be ineffective, or (2) predicting the alarm

system behavior, and use a defacement instance that will stay below a criticality level requiring human intervention.

In order to frame these considerations in a more precise context, we will use the classification of adversarial learning proposed by Barreno et al. [21], and apply it to defacement detection as follows:

1. Defacement detection is exposed to *exploratory* attacks. In fact, it may be difficult to inject examples, as this would amount to causing real defacements, the very goal our adversary is seeking. Our adversary will likely be limited to an exploratory approach, observing application evolution and content over time.
2. For defacement detection we address *targeted* adversarial models. In fact our adversary has a precise, targeted, goal: inserting a defacement that will be noticed, not just any change in a Web page. For example, a change in one long text page, where just one word is deleted, cannot be considered as a successful attack.
3. Defacement detection faces *integrity* attacks. If even just one defacement goes undetected, our adversary wins. On the other hand, it would be difficult to make the whole detection system unstable, as injecting training data (defacement examples) has a high probability of being detected, causing the used vulnerability to be fixed, and preventing further adversary action.

In summary, defacement detection should be addressed in an exploratory, targeted, and integrity context. Similarly, defacement detection naturally falls within the framework of evasion attacks that are performed at test time [37], and we will generally speak of anomaly detection with passive adversaries, who will not be able to manipulate examples in the learning set.

In fact, for adversarial applications like defacement detection, the main objective of our adversary is to predict the learned classifier. The adversary will then attack at test time with an anomalous vector that will escape detection, and yet represent a meaningful and substantial change, that will achieve some practical purpose. In the case of defacement, the adversary will simulate or predict the learning phase, and then deface the application while avoiding an above-threshold alarm to be raised.

What we then want is a learning system that is unpredictable. One way to do this would be to hide all relevant information, or even the presence of an adaptive component—a form of "security through obscurity". A more mature approach, suggested for example in [6,18,21–26,38], would be to keep the learning system public and visible, while keeping some system parameters secret.

We continue along this line of reasoning, and extend it in the context of a more general *keyed learning* methodology. A learning system takes as input training data, a feature set, hypothesis space restrictions/preferences, and other forms of bias including rule-based prior knowledge [39]. A keyed learning algorithm will take an additional input: a secret key k (see Figure 1). Round arrows indicate that the process may be repeated, as a form of online or reinforcement learning. The key and the other input data may then change over time and for different learning sessions. The key used in one learning episode will then have the role of what would be called a *session key* in cryptographic applications.

This information can be considered as a key used in symmetric cryptography, and will be known to the anomaly detection system, but not to our adversary. The learning system can use the key to:

- unpredictably select feature subsets, restrict the hypothesis space, or specify other forms of bias;
- unpredictably choose subsets of learning examples;
- unpredictably choose time of learning, time constraints on learning phases, and the amount of available computational resources.

How does the key practically influence the learning process? Starting from a key k, we have to determine a possibly large number of choices, including a specific selection of features, parameters, and input data. The number of bits in k might not suffice.

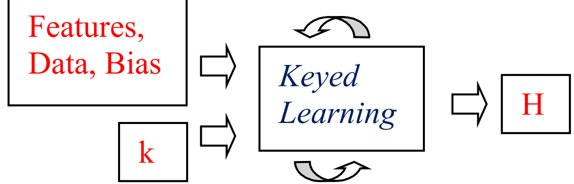



**Figure 1.** Keyed Learning, producing a learned hypothesis H, e.g., a classifier or a cluster.

We have used straightforward method, with a 128-bit secret key k, and fed it into a pseudo-random function PRF to obtain an infinite and unpredictable bit stream. A PRF may also require as input an additional seed value, a bit vector, that is not necessarily required to be secret. A number of techniques to achieve this goal have been proposed [40–42], and we have used a simplification of TLS-PRF, as defined in RFC 5246 [42]:

PRF(k, seed) =
HMAC(k, $A_1$ || seed) ||
HMAC(k, $A_2$ || seed) || ...

where || represents bit vector concatenation and $A_i$ is defined as:

$$A_0 = seed, A_i = HMAC(k, A_{i-1}) \tag{1}$$

We now have an arbitrarily long vector of secret bits PRF(k, seed). From this we draw in sequence three bit vectors: (1) bits used to select features and feature parameters, (2) bits used to select training examples and (3) bits that affect system timing and other implementation-level parameters.

Features are associated to one bit b in the first part of PRF(k, seed): if b = 1 the feature is selected and will be used during the learning phase, if b = 0 it is excluded. If a feature requires parameters (e.g., $f_j$(n) = number of lines longer than n characters), it is associated to one bit b in PRF(k,seed), used for selecting/deselecting the feature, and a fixed number of subsequent bits, representing the numeric value of the parameter n. This is all illustrated in Figure 2, while the precise use of the system timing bits will be discussed later.

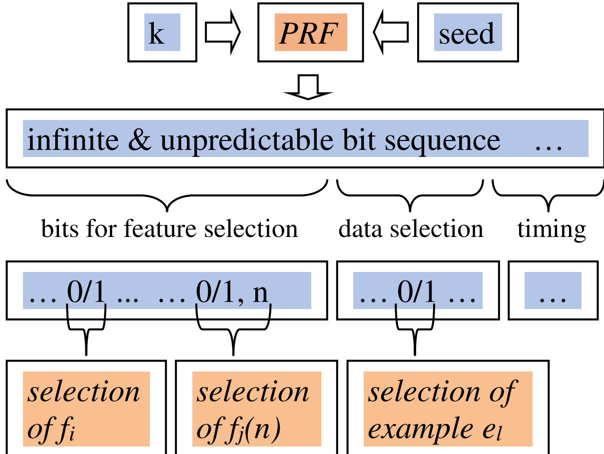



**Figure 2.** Computing the learning system's parameters from the secret key k.

*Methodology Motivations and Limitations*

In this subsection we discuss why the keyed learning methodology is needed to improve classifier performance in an adversarial setting. We then list some limitations of the approach.

The use of a secret key as an input to the learning algorithm amounts to *hiding* information from our adversary. Previous work on keyed intrusion detection [21–25] and on adversarial learning

with randomization [18–20] is based on similar considerations. The principle has also been informally suggested in the context of defacement detection [6]. In this paper, we take a security perspective, following Kerckhoffs' principle and putting all hidden information into one secret key $k$. Moreover, we extend this information hiding principle to the whole system behaviour, including aspects that were not previously considered, such as the learning and application time of the anomaly detector. The requirement for $k$ is not *randomness*, but rather *practical unpredictability*. For example $k$ could not be physically random, because it is output by a pseudo-random number generator, but, if our adversary cannot replicate the process, it will be perfectly adequate for our purposes.

In the context of anomaly detection, one may argue that use of a key or other more informal ways of hiding information from our adversary is not only a sound approach to learning with passive adversaries, but even a mandatory requirement when all input data is public. In fact, if the learning system is completely known and the input data is available, an adversary could simply replicate the learning process, obtaining the very same classifier we will use. This is the case for defacement detection, because the input examples are taken from the target Web site, and from known defacement examples. In other applications, such as anti-spam filters, the situation can be different because the examples of non-spam emails for a specific user are normally not available to the adversary. When the anomaly detector is applied in the production environment, the adversary will look for an attack pattern that is classified as normal, and use it to defeat the alarm system. As a consequence, evasion is simple and straightforward when no information is hidden from an adversary, and keyed learning is a natural solution that improves classifier performance in this context.

However, a number of limitations of the approach should be considered.

First, we must consider consequences on the prediction performance of the learning system. As stated by Biggio et al. in [18]:

> *«an excessive randomisation could lead to selecting a classifier with a poor performance, to the extent that the advantage attained by hiding information to the adversary is lost.»*

Similarly, it should be noted that the key is random and unpredictable, and could lead to the selection of a poor set of features, from a prediction perspective. The feature subset selection process induced by the key should not be confused with "feature selection" as addressed in Pattern Recognition and Machine Learning, where a set of features is selected at training time, with the goal of achieving better prediction. Here we only mean that some features are used and others are not, based on the value of the secret key—the selected set of features may not be optimally performing, and could actually be redundant and include useless features from the point of view of prediction. Nothing prevents the training phase from doing some further feature selection, in its classical meaning, after the keyed feature selection has been done. Traditional feature selection, aimed at better accuracy, is normally predictable, and thus a possible vulnerability [43] from an adversarial point of view.

As a possible mitigation of this issue, we may observe that this situation can be detected at learning time, e.g., via cross-validation with a testing set. If this happens, a new session key could be generated, and the process would be repeated. Computational effort and convergence issues should however be addressed.

As a second limitation, we cannot exclude the possibility that our adversary obtains a close enough classifier even without knowing the hidden information, for example, by relying on a large set of available data. In fact, keyed learning will hide the hypothesis space and other training parameters, but our adversary could use a different hypothesis space to learn, possibly with different methods, an equivalent or similar classifier. This issue could be studied in a PAC-learning context: for example, knowledge of the key would allow us to use a learnable hypothesis space $H_2$, while our adversary, who does not know the key, is faced with a larger, not learnable, space $H_1$. When the key k is known, the learning process will then produce, with high probability, a classifier C belonging to $H_2$, that has adequate accuracy. On the other hand, our adversary should be unable, either directly or by using another key k′, to identify a learnable hypothesis space $H_3$ that also includes C or other

accurate classifiers. In our system we have used decision tree classifiers, as they are considered to be computationally hard to learn [44]. This is illustrated informally in Figure 3.

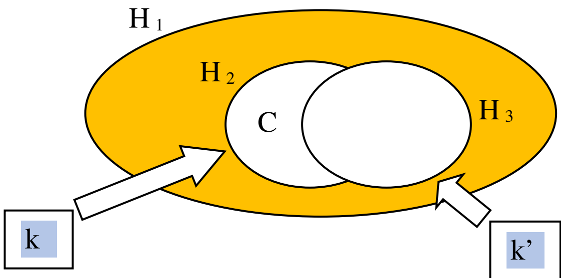



**Figure 3.** Keyed Learning—the correct key k helps select a hypothesis space $H_2$ that is easier to learn, including a predictive classifier $C$.

As a third limitation, an adversary could observe system behavior to guess the key value, possibly combined with a brute-force approach (*key recovery*). Previous work has addressed this issue in the context of Intrusion Detection [22,35], but further analysis is needed, applied to the present generalized methodology.

Finally, one could argue that the key has limited impact, because, as stated in [18], *«in real tasks the adversary hardly ever has complete knowledge about the classifier decision function»*. However, in security applications, one should not rely on general lack of knowledge by an adversary, and hide information in a systematic and sound way, normally with a key—again an instance of Kerckhoffs' principle.

## 4. System Architecture and Implementation

In the implementation of our defacement response system, we have taken into consideration the following:

- Application and web page analysis can lead to a high computational cost, and to some bandwidth constraints—this is often underestimated, but our experiments have shown that for enterprise-level Web Sites even just the scanning of all content with reasonable frequencies can be a demanding requirement.
- The analysis of a complex application or Web Site is suited for parallel processing, as there is little or no correlation between pages or site sub-directories when defacement detection is our purpose (at the state of the art [1,26], even though attacks can be massive and affect most Web Site pages, each page is individually targeted, and defaced content is not spread over multiple pages). Different pages (URLs) in the target Web Site will normally have different importance, and will thus be assigned a *weight*—for example, the home page will normally be assigned the highest weight.
- The implementation should be suited for installation with multiple clients with different and difficult to predict IP addresses, so as to avoid differential defacement [14], i.e., sophisticated defacement instances where normal page content is returned to clients that are suspected monitors, whilst defaced content is returned to normal end-user clients.

As a consequence, our implementation addresses one page or application component at a time, in parallel and from different IP sources.

### *4.1. Keyed Scheduler*

As a first step, we will analyse the concept of a "keyed scheduler", i.e., a system component that will spawn a process and determine its inputs in a way that is unpredictable for our adversary. This will be an essential component of our defacement detection architecture, and is described in Figure 4.

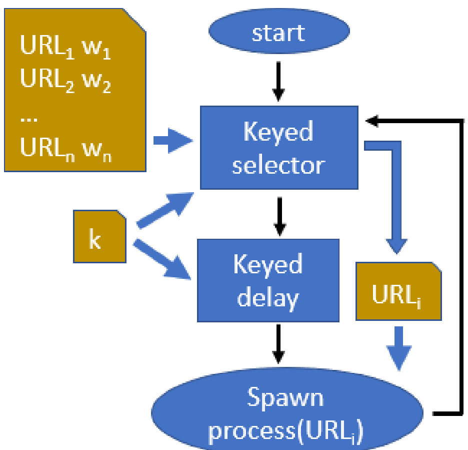



**Figure 4.** Keyed scheduler.

The keyed scheduler will use as input a list of URLs and corresponding weights:

$$< URL_1, w_1 >, ..., < URL_n, w_n > \quad (2)$$

as well as the secret key $k$. It has two components: a keyed selector and a keyed delayer.

The keyed selector will select one URL out of the input list. The weights associated to each URL determine the frequency of their use:

$$frequency(URL_i) = w_i / \sum_{1 \leq j \leq n} w_j. \quad (3)$$

The choice of the URL will depend on the key $k$ and should respect the above defined frequencies. As a first step, a frequency table $FT$ is constructed:

$$FT = [[URL_1, 1], ..., [URL_1, w_1], ..., [URL_n, w_n]] \quad (4)$$

where each $URL_j$ is repeated $w_j$ times. The length of $FT$, that we will call $m$, is then computed as

$$m = \sum_{1 \leq j \leq n} w_j \quad (5)$$

The keyed selector's output $URL_i$ will be computed as $URL_i = FT[index][0]$, where

$$index = HMAK_k(A_t) \quad mod \quad m \quad (6)$$

and $A_t$ is obtained as part of our pseudo-random function PRF as defined in Equation (1), computed for the current iteration $t$. The value of $t$ will be incremented for every new cycle of the keyed scheduler, i.e., every time the keyed selector is activated.

In summary, $index$ will be an unpredictably chosen element of $FT$, and thus $URL_i$ will be chosen unpredictably out of the input list, respecting the given frequencies. The user will then be able to assign different weights to the URLs in the list, depending on their relative importance.

The keyed delayer will wait for an unpredictable time interval T, measured in milliseconds, where $T \leq MaxTimeInterval$:

$$T = HMAK_k(A_t) \quad mod \quad MaxTimeInterval \quad (7)$$

The adversary will then be unable to predict this time interval, and hence know when the detector will be running.

After the keyed delay, the keyed scheduler will spawn a process and pass to it the selected URL. The process will run on a new selected client. Again, the client could be selected based on the secret key $k$. In our implementation we have used a simpler round robin strategy, so that all available clients are equally used. The resulting effect is a monitoring process being spawned at unpredictable times and with an unpredictable input URL.

### 4.2. Overall System Architecture

The overall system architecture is described in Figure 5, where all activity is triggered by two separate keyed schedulers: (1) a scheduler for the URL tester, on the top right side of the figure, and (2) a scheduler for the keyed learning component, at the bottom of the figure.

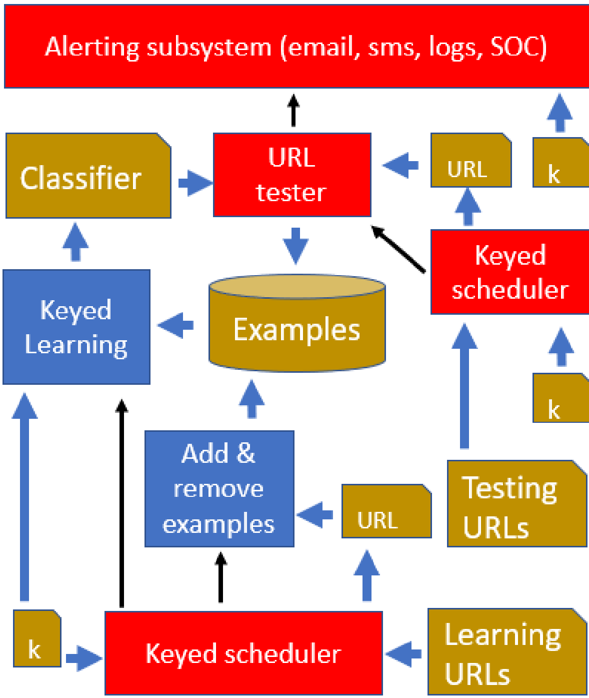



**Figure 5.** System architecture.

The input to the scheduler for the URL tester is a comma separated list of values (CSV), where each value consists of a <URL, weight> pair. We generate the URLs in the CSV by crawling the application, while corresponding weights decrease as we run further from the site's document root, unless our application-specific component risk labeling states otherwise. Thus, in general, the application's home page will be checked more frequently than content leaves, that may have a lower reputational impact.

The scheduler will spawn a process in due time, and a corresponding URL testing procedure is followed (see Figure 6). As a consequence, also within the same client, multiple <URL, weight> pairs are consumed by independent processes running in parallel.

First, the URL content is downloaded, together with its inline external components (scripts, images, etc.). In our experimentation, we targeted a large enterprise site where a CDN was in place. In this case, it is important to download content from the original content source, and not from the CDN or intermediate proxies.

Second, two basic tests are performed, availability and equality (see again Figure 6):

- **Availability**: if the URL cannot be downloaded, because an error is returned (e.g., http 4xx or 5xx), or a timeout occurs, the availability test fails and the spawned process instance stops. A corresponding log line is stored and a possible alarm is generated in the alerting subsystem.
- **Equality**: if a hash of the downloaded content is equal to the one computed for previous downloads, the detection procedure was already run with the same data, and again the process

stops. This is very important in practice, because the equality check will cause a stop most of the time, as the application does not change continuously, resulting in significant computational savings. Equality is actually verified in two steps:

– domHash: compares the hash (sha256) of the most recently saved DOM, with the DOM that was currently downloaded;
– contentHash: extracts the "pure text" from the current DOM (including the *alt* tag values), and from the previously saved DOM, and checks the corresponding hash values for equality.

If domHash succeeds (the two hash values are equal), then equality succeeds and the process stops. It may, however, happen that domHash fails only because some script names differ or other content-unrelated differences apply—in this case contentHash succeeds and the defacement detection process stops, without generating any alarm. We do the check in those two phases for efficiency reasons, as domHash is more easily computed and detects most situations that do not require further processing.

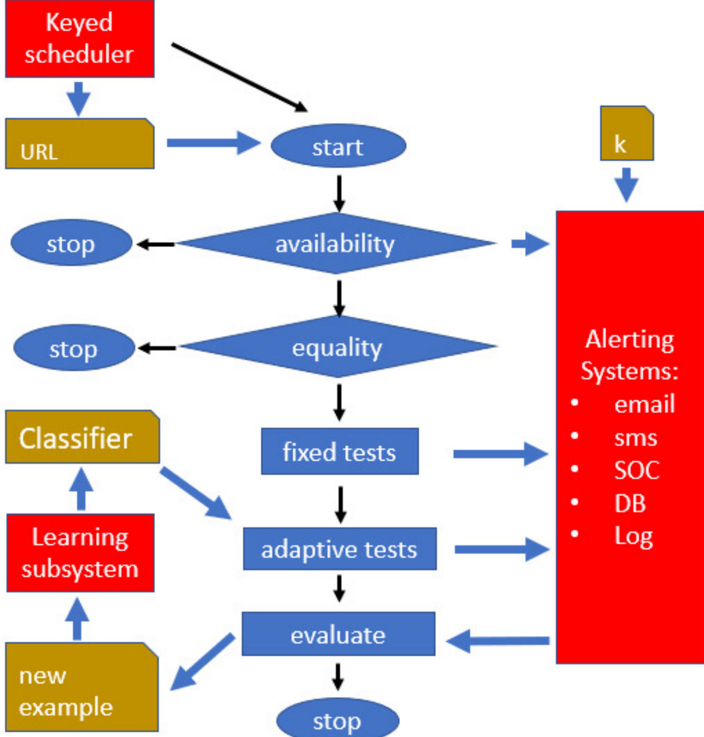



**Figure 6.** URL tester.

Third, a number of fixed custom tests, and one adaptive test are fired, yielding an overall defacement score. This will cause possible logs, alarms and SOC-alerting procedures to be generated.

The overall system is then composed of three functional components: (1) keyed schedulers, (2) keyed learning, and (3) URL tester. The components are interconnected as shown in Figure 5, and the motivation of each of them can be summarized as follows:

- There are two keyed schedulers: a "testing" scheduler (top and right in Figure 5) and a "learning" scheduler (bottom of Figure 5). The schedulers work as shown in Figure 4 and have the purpose of making system actions start at times that are unpredictable for our adversary. The use of a testing scheduler is a common requirement of all complex alarm systems: if the adversary knows when the system checks for anomalies, the time of attack will be planned so as to avoid the check. For similar reasons we also make the learning time unpredictable, so that the adversary will not know when the classifier undergoes possible changes.

- The keyed learning component (middle and left in Figure 5) is of course essential in the system architecture, because it is responsible for producing a good classifier, based on available examples, prior knowledge, and the key $k$. The system would not adapt to Web site changes without this component, and it would be deterministic and predictable, so the component is essential for classifier performance (UDR, FAR and unpredictability as defined at the end of Section 2.1).
- The URL tester (middle and top box in Figure 5, details in Figure 6) is spawned at unpredictable times and with an unpredictable input URL—it will check whether the corresponding Web page has undergone a possible defacement, based on the currently available classifier. So this component is essential and necessary for generating possible alarms, as required in defacement detection (see again Section 2.1).

### *4.3. Basic Custom Tests (Fixed)*

The fixed tests are encoded custom tests, not resulting from a previous training phase. We soon realized this was necessary because in practice some simple, common sense checks are extremely effective. For example, just checking the existence of the correct company logo in the correct position detects a large number of real-life defacement instances.

We will label these custom tests as $T_1, ..., T_n$, and they are part of the "fixed tests" box in Figure 6. Some tests are binary (with 0 or "fail" meaning a possible defacement and 1 or "success" meaning a possibly normal content), other tests are weighted. The implemented tests may include the following:

- dnsSpoof: given a list of IP addresses (or ranges of IP addresses specified with CIDR notation), it verifies that the response comes from a "trusted" IP address, belonging to the given list
- untrustedSources: given a list of trusted domains, it verifies that scripts or resources from external and untrusted sources are not included in the page
- publicDefacedSitelist: verifies that the given URL or its domain component are not present in the RSS feed of Zone-h.org, containing the list of the last 20 "important" sites that have been defaced
- imageNumber: compares the number of images in the previous and in the current DOM—if there is a difference of more than a given threshold (defaulting to 10%), the test fails
- lineNumber and wordNumber: same as the previous one, but the number of HTML lines and words is compared
- language: identifies the prevalent natural language in the downloaded content, and verifies that it is within a pre-configured list of accepted languages (e.g., English, Italian)
- wordBlacklist: given a word dictionary, divided in subcategories (e.g., hacking, spam, politics), it counts the number $N_u$ of such undesired words or word variants in the page, and returns a weight $w = N_u/N_t$, where $N_t$ is the total number or words in the page
- diffDistance: using a sequence matching algorithm, the difference between the current and the previous DOM is computed (resulting in a floating-point number between 0 and 1, with 0 meaning equal, and 1 maximally different)—this is then combined with the site's changing frequency to yield a test output in the form of a numeric weight between 0 and 1
- diffOCR: starting from the page's screen shot, an OCR routine is run yielding an alphanumeric sequence; this is again processed as in the diffDistance test, yielding a difference weight between 0 and 1
- dictOCR: the words obtained via OCR are processed as in the wordBlacklist test
- fixedImage: a specific image in a specific position must occur (e.g., company logo)
- fixedText: a specific text sequence in a specific position must occur (e.g., company motto)
- applicationSpecific: other custom tests have been implemented that are specific to the target application.

Some of the above tests are assigned a weight and are combined to yield a "fixed test weight".

It is important to note that the computation of the above tests provides us, as a byproduct, with a number of computed quantities that will be useful for the following adaptive test. For example,

the lineNumber test computes the number of lines, and the dictOCR test produces a list of words extracted from page images. These quantities are used as precomputed features in the adaptive test, avoiding duplicate evaluation of the same quantities. This is also why, in Figure 6, the adaptive test follows the fixed tests, and receives input from them. The fixed tests can run in parallel threads or processes, but the adaptive test will have to wait for all such processes to terminate before it can start execution.

### *4.4. Adaptive Tests (Based on the Learned Classifier)*

After the fixed tests $T_1, ..., T_n$ have been executed, an adaptive test, based on the learned classifier, is activated. We will label the adaptive test as $T_L$. This step requires input data from $T_1, ..., T_n$, because most of these custom tests will have computed quantities that $T_L$ needs to use as features. In this way $T_L$ will not have to recompute such values (e.g., $T_i$ is the above-mentioned test wordNumber, verifying differences in the number of words: it will have computed the number of words in the Web page, and $T_L$ could need it as a feature value to be used by the learned classifier).

$T_L$ is fully defined by the features to be used, the selected positive and negative examples, and the keyed learning methodology, as described in the following subsections.

#### 4.4.1. Feature Choice

Following our previous discussion on keyed learning, we face the goal of identifying a large number of features, where only a subset of such features will actually be used for training. The choice of this subset will depend on the learning key, that is selected at deployment time, can be changed at run time, and is unknown to our adversary.

The features to be used should be adequate for addressing the defacement problem, and hence should be

- easy to compute for general Web pages, and preferably available from previously computed values in the custom tests $T_1, ..., T_n$
- applicable to dynamic pages and pages containing different types of client-side code
- adequate for describing “normal” page content and “defaced” page content
- capable of capturing the page look and feel
- easy to extend and change, so as to allow for larger keys

Based on the above guidelines, and using some of the features also suggested by [6,7], we have identified a large number of features, including:

- Global page features, e.g.,
    - Empty space to Filled-in ratio
    - Prevalent colors
    - Text to image ratio
    - Static to moving content ratio
    - Amount and type of client side code
- Simple numeric features, e.g.,
    - Number of lines
    - Number of characters
    - Number of images
    - Number of images from external URLs
    - Number of hyperlinks

- Words and semantics, e.g.,
  - Number of words in a given semantic classes or dictionaries
  - Prevalent languages (e.g., 1st English, 2nd Italian, 3rd French)
  - Presence of mandatory words / lines (e.g., company motto)
- DOM tree
  - Depth, arborescence
  - Number of leaves and nodes
  - Tags
  - Fonts
  - Character size and color
- Images
  - Color map (e.g., RGB mean in given areas)
  - Average image size
  - Prevalent image position within page
  - Presence of predefined images in a given page area (e.g., company logo)
- Text from image OCR
  - Word to size ratio
  - Prevalent semantic class in image words
  - Total number of words in images
- Network data
  - Download time
  - Number of different IP addresses contacted
  - Number of TCP connections

Some of the above features can easily be changed or extended, so that an even larger number can be produced. Some can even be parameterized, for example number_of_links(k,n) is defined as the number of external links in the k-th portion of the page, after dividing the page in n segments of equal size.

#### 4.4.2. Keyed Learning Application

As explained above, and highlighted in Figure 1, a secret key is used, and initialized when the anti-defacement system is deployed—it may be changed later and will be stored and known on this system only. With this key, we will select examples, features, and learning method and parameters, as discussed in the previous section on keyed learning, and illustrated in Figure 2. The following uses of the key are included.

- Keyed selection of examples

  An initial set of positive (acceptable page content) and negative (anomalous, possibly defaced page con-tent) examples is obtained as follows:

  - Positive examples: a reference version of the target URL (validated by the end user), system-generated modifications of this page, previous versions downloaded from the same URL, pages downloaded from the same Web Site, pages from a defined set of related URLs.

    - Negative examples: system generated modifications of the target URL's content, examples of defaced pages (e.g., from zone-h), pages for a defined set of URLs with content considered as inappropriate, content from applications that are semantically distant from the target.

  For all of the above examples, the complete set of features is computed and their value is stored. A sub-set of positive and negative examples is selected based on the secret key. Examples may be weighted, where a higher weight gives an example a higher probability of being selected. For instance, the positive examples from the target URL may have a higher weight, with respect to the ones obtained from related Web sites, and high-impact defacement examples in the style of "hacked by anonymous" could be given a higher weight based on risk evaluation issues. Such weights are considered as domain knowledge and are not part of the secret key.

- Keyed selection of a subset of features

  The secret key is used to select the features that will actually be used, among the complete set of available features. The keyed selection of features may include the setting of parameter values in features that include this option (e.g., selection of k and n in the above-mentioned feature number_of_links(k,n)). Features with parameters are selected and parameters are set as described in Figure 2.

- Keyed choice of the hypothesis space, the learning algorithm and its parameters

  The secret key is used to define a general hypothesis space, e.g., decision tree versus Boolean formulae. A subspace may also be defined with parameters, e.g., kDNF or k-term-DNF for Boolean formulae. Other parameters, also selected based on the secret key, may be used to define or constrain the learning method, e.g., minimum number of examples covered by a decision tree node, or pruning criteria. Any form of learning bias should also be codified with components of the secret key. This is not explicitly described in Figure 2, but an additional and unpredictable bit vector could be generated for addressing such forms of algorithm and bias selection.

- Keyed selection of learning times and schedule

  The secret key will also define when learning takes place. As for the URL defacement test and alarm process, the learning schedule will also be defined with a secret, comma separated list of URLs and weights (see Figure 5). When the keyed scheduler in the bottom of the figure outputs a given URL, a process is spawned and will perform a new learning phase, and the example database will be updated. In fact, the learning phase will not be done only once at the beginning, because:

    - The secret key may change. Actually, it will be a good idea to maintain a master key, and generate "learning session keys" to be used for a given period of time, so as to prevent our adversary from inferring secret information based on the observation of generated alarms and site management actions.
    - The examples might change. When the URL content is willingly modified by the site owner, it will make good sense to add the new version as a positive example, possibly with variants. In the same way, if new defacement examples become available, we want to add them to the negative examples.

  It is important to hide the time when a new learning episode is triggered, so as to prevent the adversary from guessing possible classifier behavior. The "learning URLs" CSV (containing pairs <URL, weight>) and the secret key k constrain the possible moment when the learning process is spawned and the URLs it will use. The same is done for making the moment of defacement testing unpredictable, in the top of Figure 5—in this case the "testing URLs" list is used.

After the learning phase has completed, we have a new classifier C, that will label any new URL content as either normal or anomalous. We have in fact used a binary classifier (yielding

a normal/anomalous classification), but multiple classes (e.g., normal/suspect/anomalous) or regression could be used, yielding a numeric evaluation of the degree of anomaly. This classifier C is then input to the "adaptive tests" box in Figure 6.

### *4.5. Combined Evaluation and Alerting*

The outputs of $T_1, ..., T_n$ and of $T_L$, are finally used in a combined numerical evaluation (hidden in the Alerting Subsystem of Figure 6): based on defined thresholds, that depend on domain knowledge and on the secret key $k$, we produce an overall alarm level: No_alarm, Information, Warning, and Critical.

"No_alarm" will cause no action. "Information" and "Warning" produce a log line, with corresponding labels and parameters. "Critical" will produce a log line, a web service communication to the SOC, and a message (email or sms) to a human operator, as defined by our service team configuration. The intervention of a human operator will provide, as a side effect, a validated classification of normal or defaced content, that will be added to positive and negative examples, respectively (box "evaluate" in Figure 6).

### *4.6. Software Tools and Platform*

We have deployed a testing platform and system, as well as a production platform; the testing platform is based on the above-described functionalities, and uses the following software and platform components:

- Platform: Debian 8 jessie Kernel, version 3.16.0-4-amd64
- Software and libraries: Python 2.7, running in Virtualenv
- DOM reading: Phantomjs 2.1.1 with Selenium Web-Driver
- Learning System: scikit-learn library
- Cryptographic libraries: OpenSSL 2.0.13
- Alerting method: Email and SMS in case of critical events, Unix-style logging files

The testing system runs in a virtual machine with 1 GB RAM and two i5 Cores, and the production systems use 1 GB and 2 E5606 @ 2.13Ghz Cores.

## **5. Results: Experiments, Full-Scale Deployment and Discussion**

We will describe three different experiments:

- (Section 5.1) a newspaper scenario, where a real web site has been used, but simulated changes and defacements were injected
- (Section 5.2) the same newspaper scenario, where defacements were injected by simulating an adversary, in two cases: (1) when the adversary does not know the key $k$ used for learning and (2) when the adversary knows the key
- (Section 5.3) a production environment where our system is presently being used as a defacement response component.

This approach was chosen because no actual defacements were expected to happen in the production environment. As a consequence, some performance parameters could only be evaluated in the newspaper scenarios.

### *5.1. Newspaper Scenario*

#### 5.1.1. Setup

The test system has been used to monitor 10 selected URLs of a major news website. The URLs were chosen based on the likelihood of changes and expected update frequency.

To simulate an attack, we used the Nginx web server, as a virtual host that responds to local requests. Nginx will respond to HTTP requests sent to the local domain with a PHP page that loads content from a substitution source, locally hosted. This substituted content can be (1) a modified version of the expected page, (2) a page from another news website or (3) a known defacement instance from Zone-H [16].

Before some randomly chosen executions of the defacement detection tool, a script will add a line to the local *hosts* file, in order to redirect HTTP requests to the local virtual host. On those random occasions, we will thus obtain locally substituted content, instead of the remote target page. Afterwards, the script deletes the line from the *hosts* file, to restore normal operation. In this way, we use the modified page for virtually just one execution of the tool. When defacement detection is performed with a high frequency, the modified page can be found for more than one execution.

A Python script runs daily, decides when to carry out the replacement of the page, and generates a configurable number of tasks, to be scheduled randomly during the day. The defacement examples used in the replacement are not used in the Machine Learning phase, in order to make the test as realistic as possible.

#### 5.1.2. Performance and Results

To assess the accuracy of the tool it is useful to evaluate its behavior in the context of the response mechanisms as previously discussed, where our combined evaluation produces an overall alarm level: No_alarm, Information, Warning, Critical. We also consider this experimental setting, where two possible cases are at hand: (1) the original URL content is downloaded or (2) a replacement page is obtained. The replacement page can be (2.1) a defacement instance D, (2.2) a page R from a similar Web Site or (2.3) a modified version R of the same page. We then found it useful to define three kinds of errors:

- False positive: there has been no page replacement of any kind, as defined in case (1) above, but the system evaluated the URL as critical.
- False warning: there has been no page replacement, again as defined in case (1) above, but we obtained a warning because an ordinary change in the original page was detected.
- False negative: there has been a page replacement including some defacement instance D, as in case (2.1) above, but the evaluation is either "Information", "No_alarm", or "Warning"—since this is a defacement we would have wanted a "Critical" evaluation.

The percentage of false positives corresponds to the *False Alarm Rate* (FAR) as defined in Section 2.1, and the percentage of false negatives corresponds to the *Undetected Defacement Rate* (UDR).

We have run the system over a three-day period, injecting a total of 150 defacements, obtaining the results reported in Table 1.

The results are also represented graphically in Figure 7. The defacement detection system was run every 10 minutes during the three-day period. For every execution, three quantities were measured:

1. The total number of "Critical" alerts that were generated for the 10 URLs being monitored
2. The total number of "Warning" alerts that were generated for the 10 URLs being monitored
3. The value of the combined numerical evaluation computed in the alerting subsystem, averaged over the 10 URLs being monitored.

The above three values were thus obtained every 10 minutes, yielding a total of 174 critical messages and 272 warnings over the three days. The average evaluation value and the total number of warning and critical alerts over one hour periods are plotted in Figure 7. Since there is a total of 10 URLs, a maximum of 60 alerts can be obtained every hour ($10 \times 6$ executions per hour). The graph of Figure 7 highlights four semantically meaningful events that occurred during those three days, as reported in Table 2.

**Table 1.** Accuracy in the newspaper scenario.

| Number of Defacements | False Positives (FAR) | False Warnings | False Negatives (UDR) |
|---|---|---|---|
| 150 | 10 | 266 | 1 |

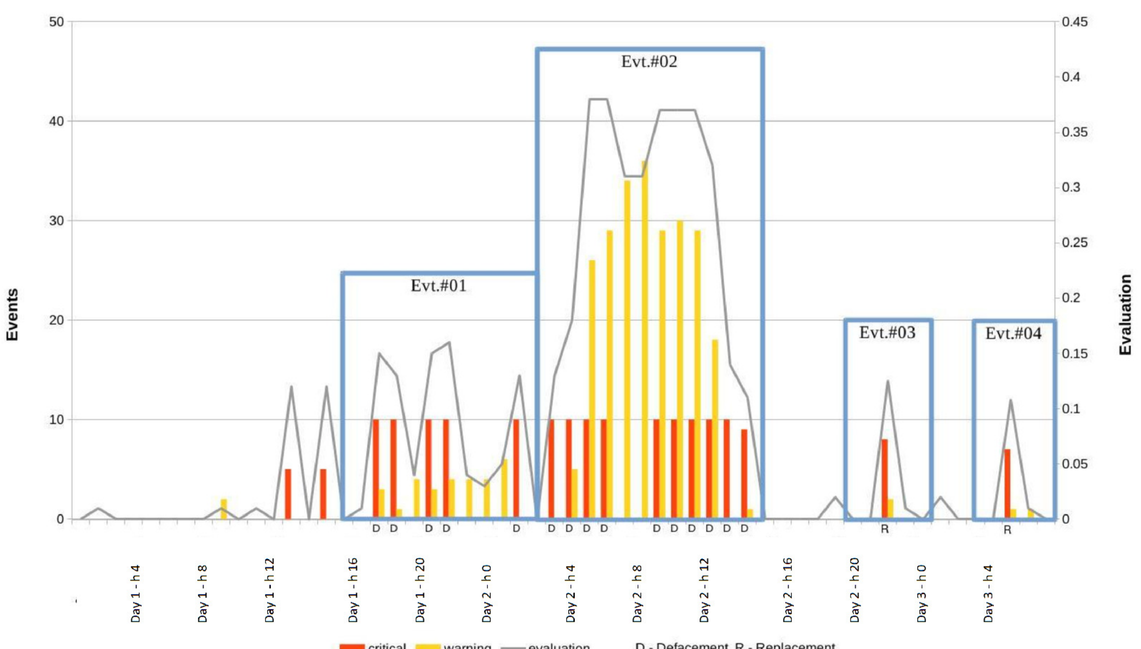



**Figure 7.** Aggregated evaluation and relevant events (newspaper scenario).

**Table 2.** Major events in the newspaper scenario.

| Event | Description |
|---|---|
| Evt.#01 | Redirection to a local defacement page |
| Evt.#02 | Warning events due to the presence of an article describing a "hacking" incident that had just made the news, plus some of our injected defacements. Moreover, some artificial defacements were also substituted, as in Event #01. |
| Evt.#03 | Replacement with a Web Site from another newspaper—there are big differences in the structure and the contents of the site, with respect to our target URLs |
| Evt.#04 | Replacement with a different page from the same target Web Site |

Taking into account the dynamic nature of the target Web Site, the experiment produced the expected results. We obtained a moderate number of false positives and a low number of false negatives.

As reported in Table 1, a total of 266 false warning were obtained—these do not necessarily require human intervention and can be easily dealt with in a running installation. They are due to significant changes on the target pages, as is it common for a news web site. This can include major text and keyword changes, as well as abrupt changes in the page size, text to image ratio, and number of images.

The learned component of the classifier produced no False Alarms (FAR = 0%), and an Undetected Defacement Rate (UDR) of 18%.

The overall accuracy is better for UDR, as reported in Table 1, because the global evaluation combines $T_L$ with the fixed tests of Figure 6. It is interesting to note that the result of the classification depends on the selected key k. Changing this, in terms of features and selected examples, causes an unpredictable change in the classifier. When the selected page is not a defacement from Zone-H,

but a replacement page from the original Web site or from another newspaper, a lower number of alerts is generated (events #03 and #04), as one would expect.

### 5.2. Adversarial Newspaper Scenario

We now describe an experimental setting similar to the one of the previous subsection, but where defacements are chosen by an adversary. Since the focus of this section is about studying the effect of adversarial action, only the adaptive classifier $T_L$ was used, and the custom tests previously described were bypassed.

The test system has been used to monitor just one selected URL from the same news website. A set P of "normal" page examples and a set N of "defaced" page examples were generated, where normal pages are relatively simple variants of the newspaper page, and defaced examples are obtained by injecting content from Zone-H [16] defacement content into variants of the newspaper pages. P is divided into a training set (PTr) and a test set (PTs). N is divided into a training set (NTr), a test set (NTs), and an adversarial defacement set (ADS). A random forest classifier $RFC_k$ is learned using the training set PTr ∪ NTr, after selecting a secret key $k$, based on the previously described keyed learning methodology.

To simulate an attack, we suppose that when our adversary knows $k$, she can learn a classifier $RFCA_k$, starting from the same learning set PTr ∪ NTr. Assuming the adversary has access to the same training set we use amounts to giving her a lot of information. In many practical cases, the adversary does not have access to this, or we could use the key to hide it. In this experiment we have decided to simulate a powerful adversary, having access to the same training data we use. However, the classifier $RFC_k$ we have previously generated and $RFCA_k$ will not normally be functionally equivalent, because of the variability instrinsic in the random forest generation and in other system components (e.g., timeouts, hyper-parameters). The key was used in this experiment only to select the features to be used during traning.

We used again the Nginx web server, as in the previous subsection. The browser will in this case always contact the local Nginx server, that will behave in two possible ways:

1. download the real URL content from the newspaper site, and return a variant of it (then log "page tag = normal"), or
2. in some randomly selected cases, use a symbolic link in the file system, pointing to a substitution page (a defacement), return this page, and log "page tag = defaced". The link to the substitution page is changed by a script that is run repeatedly, and is chosen from the adversarial defacement set ADS. This choice can be done in two ways:
    (a) pick a random page from ADS (when the adversary does not know $k$), or
    (b) run through the ADS set and pick a page that, using $RFCA_k$, is tagged as a "normal".

After the page is returned as just described, we classify it with $RFC_k$ as either normal or defaced, and log this classification ("class = normal" or "class = defaced"). Finally, we compute FAR and UDR as follows:

FAR = | { logs with "page tag = normal" and "class = defaced"} | / | {logs with "class = defaced"} |

UDR = | {logs with "page tag = defaced" and "class = normal"} | / | {logs with "page tag = defaced"} |

In case 2(b), the adversary tries to pick from ADS a defacement page that will escape detection. More precisely, knowing the key $k$, she learns a classifier $RFCA_k$, and uses this to select a defacement that is likely to be classified as "normal" by our detector. If $RFC_k$ and $RFCA_k$ are semantically close, the adversary will be likely to achieve her goal, and cause a high UDR. The results are reported in Table 3, and the experiment can be replicated using the software and data made available in www.di.unito.it/~fpb/KeyedLearningData/.

**Table 3.** Results for the adversarial news scenario.

| Adversarial Knowledge | FAR | UDR |
|---|---|---|
| $k$ known | 11.11% | 88% |
| $k$ not known | 10.81% | 25% |

As expected, if the adversary knows the key, she can simulate learning, thus obtaining a classifier that is similar to ours. She will then pick defacements that are likely to go undetected (UDR = 0.88). If, instead, she does not know the key, all she can do is pick a random defacement, and face detection based on expected system performance (UDR = 0.25). In this setting, it is then clear that keyed learning is essential for preventing adversarial action, since UDR changes dramatically when the adversary does not know the key.

### *5.3. Production Environment*

#### 5.3.1. Setup and Context

The production environment targets a single Web Site of a large company in the industrial sector, where about 260 active URLs are being monitored by the system, as part of an outsourced security maintenance contract.

The activity started in a moment when the company was changing its Web Site completely, including graphics, CMS and content. Our defacement monitoring was started on the development environment for the new Web Site, and continued in the early stages of its production deployment—it is currently still active as the company has decided to use it as an additional and ongoing security control. This context implies that:

- The test is interesting, as there has been significant content changing activity in the early stages of the transition to the new Web Site, potentially causing a large number of false alarms (false alarm rate—FAR)
- This is a real-world test, but it cannot be used to measure the percentage of defacements that were not recognized (undetected defacement rate—UDR): in fact, as one would expect and hope, no real defacement ever occurred during the reference period. To evaluate UDR, we have then used the experiment in the simulated, testing environment described in the previous subsection.

The 15 top level URLs in the Web Site's tree are monitored with a frequency of 5 min, and the full site is monitored once per hour. The exact time when the test is performed is, however, randomized and made unpredictable by using our secret key, based on the keyed scheduler.

#### 5.3.2. Performance and Results (Deployment Phase)

We first analyze an initial period of four months, when the site was set up, tested, and filled with content. Since no real defacements have occurred, we measure the number of false positives that caused an alarm, due to their higher "evaluation" result (called "notification" events), and we count the number of critical messages and warnings. Let us start by analyzing just one month (Table 4).

**Table 4.** Single month.

| Period | Critical Events | Warning Events | Notification Events |
|---|---|---|---|
| M4 | 3 | 0 | 3 |

The three events mentioned in Table 4 are caused by a known bug in the Selenium WebDriver, that occurs when HTTPS is used: for self-signed certificates the page may be loaded incorrectly and the WebDriver returns an empty page. They caused a notification.

A graphical representation is given in Figure 8, where we report, for every day: (1) the total number of critical messages, (2) the total number of warnings and (3) the day's average for the combined evaluation of fixed and adaptive tests. The three critical messages of Table 4 are visible in the first part of the month.

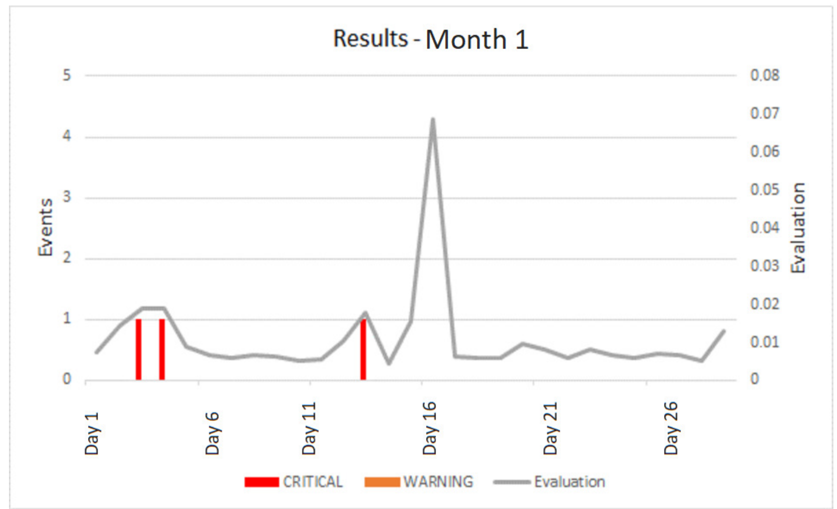



**Figure 8.** Aggregated evaluation graph and relevant events (production scenario—month M4).

On day 16, the CMS users made some changes that affected all the pages, as highlighted by a relative peak in the graph. However, every evaluation for a single URL was below threshold, so no event was triggered and no message or warning was generated.

For the complete deployment period (four months: M1, M2, M3, M4), a total of 975 events (warnings or critical messages) were generated, as reported in Table 5.

**Table 5.** Total monitoring period.

| Period | Critical Events | Warning Events | Notification Events |
|---|---|---|---|
| M1–M2 | 393 | 324 | 66 |
| M3–M4 | 258 | 0 | 6 |

Only 72 events caused a notification. Again, a graphical representation with average daily evaluation and total number of messages is reported in Figure 9. One can observe the small peak on the right end side of Figure 9, corresponding to the main peak of Figure 8.

With reference to Figure 9, we identified four major events that have a semantic relevance, where the system reasonably generated several alerts (see Table 6).

**Table 6.** Events.

| Event | Description |
|---|---|
| Evt.#01 | The development environment is targeted: most pages are empty or filled with mock content, many changes and rollback |
| Evt.#02 | Website go-live; after a CMS error (peak on month M2, day 27), content insertion began |
| Evt.#03 | Website responding with an error |
| Evt.#04 | CMS temporary shutdown |

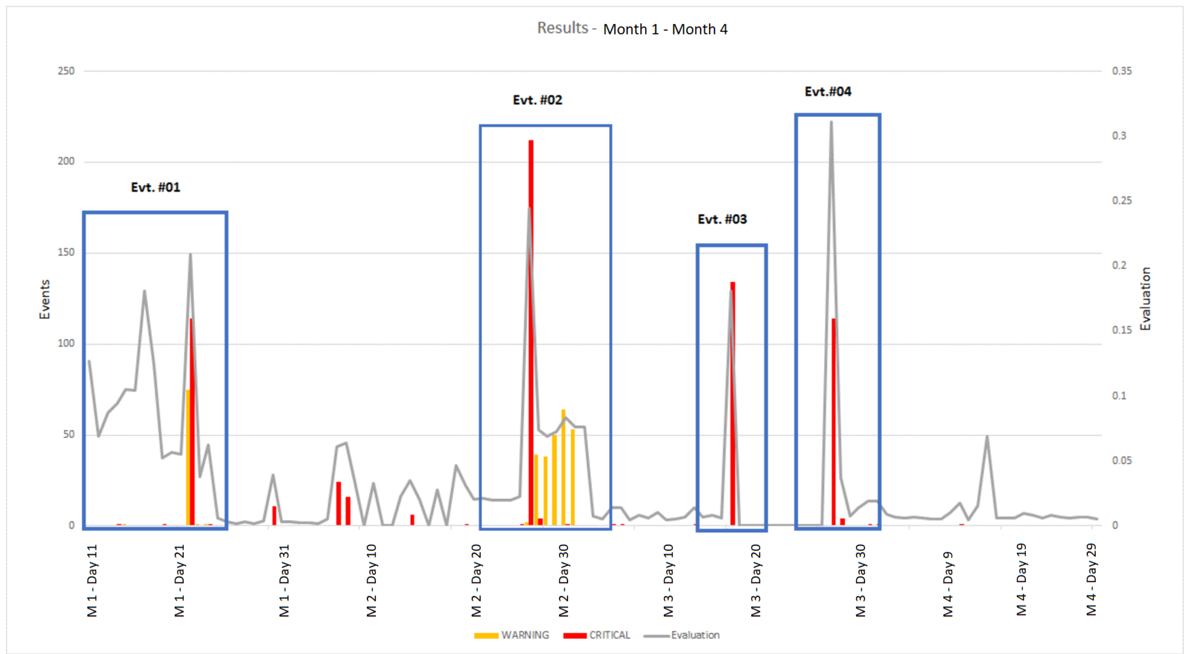



**Figure 9.** Aggregated evaluation graph and relevant events (production scenario—M1 till M4).

#### 5.3.3. Performance and Results (Maintenance Phase)

During the maintenance period (that we have called the "maintenance months", MM2 to MM9), after the Website was deployed and became stable, system-generated alerts were less frequent, as reported in Table 7 and Figure 10. This is a desired situation during normal maintenance, because with a limited number of alerts it is possible to obtain manual inspection of the corresponding pages, after a criticality is automatically detected by the system.

**Table 7.** Total monitoring period.

| Period | Critical Events | Warning Events | Notification Events |
|---|---|---|---|
| MM2–MM9 | 4 | 4 | 2 |

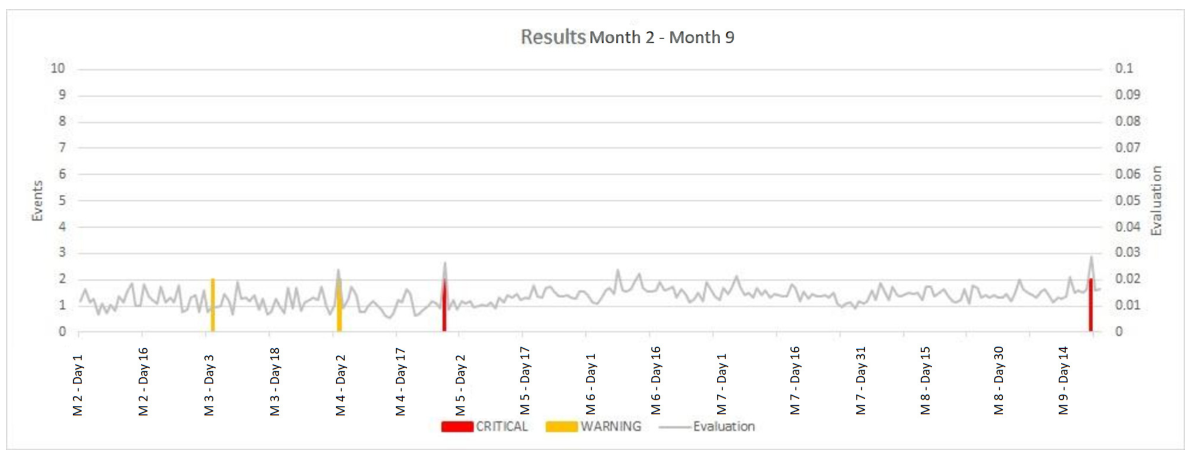



**Figure 10.** Aggregated evaluation graph and generated alerts (maintenance months MM2 till MM9).

### *5.4. Limitations*

Defacement detection is an important, but very particular case of anomaly detection:

1. it is extremely unbalanced, because in normal operating conditions all data is non-anomalous, but just one defacement instance has catastrophic consequences

2. once a defacement attempt is detected, it is game over for the adversary, because the vulnerability will be fixed and no additional attempts will be possible
3. the adversary normally has available all the non-anomalous examples, when the Web Site is public, as it normally is
4. defacement examples are available (e.g., Zone-H [16]), but they are normally unrelated to the target Web site and should not be considered as representative of possible defacements that will occur in the future in the target organization.

As a consequence, we did not simulate adversarial behaviour in the production scenario. This was done, e.g., in [18] for the case of anti-SPAM applications. In that case, the above enumerated items do not apply. Yet, a number of simplifying assumptions had to be done: (a) a limited number of equi-probable randomization choices were possible (100 different sets of weights), (b) the adversary could arbitrarily reduce the score of each anti-spam module, and (c) the cost of such reduction was assumed to be proportional to its value. In our production environment even more assumptions would have been needed, due to the particular application characteristics as enumerated above.

We did, however, apply a similar strategy in the newspaper scenario (Section 5.2), and the results show that keyed learning is effective: if the key is not known, the measured UDR is significantly lower, when compared to the value obtained when the key is known to an adversary.

In the real world application context described in Section 5.3, our detection component was integrated with a SOC service and a CDN. We measured success based on FAR and UDR, and modeled the presence of an adversary as a *constraint*: the adversary should have as little information as possible about the learning procedure and the overall detection system. In fact, an adversary could avoid detection if this information were available. Moreover, we wanted to avoid so-called *security through obscurity*, and all hidden information is derived from an $n$-bit secret key, where $2^n$ choices are possible. As a consequence, not only does the adversary lack information, as in the case of the introduction of a limited number of randomization choices, but she will also face a computational difficulty, as not all key values can be feasibly tried.

A question remains unanswered: could an adversary predict classifier behaviour by some other means, i.e., without trying all possible key values? This is an intriguing area for future work. Moreover, we suggest that keyed learning should be applied to other areas of anomaly detection, where it is necessary to hide classifier details in a way that is both systematic and unpredictable.

## 6. Conclusions

The present study is, to our knowledge, the first systematic application of adversarial learning to defacement detection, although the concluding sections of the article by Davanzo et al. [6] had previously addressed the issue, advocating the need to hide learning parameters in this particular application context.

A keyed learning methodology was implemented and integrated in a real-world defacement response system. The obtained results can be considered successful from two point of views:

- *practical usability*: the number of alarms was very limited during the maintenance phase, and still acceptable even during the content insertion and set-up phase;
- *security*: in the production scenario, all significant changes in the contents and graphics were detected and generated an alarm. Moreover, in the partially artificial newspaper scenario, all injected defacements were recognized, and keyed learning was shown to be effective to limit adversarial success.

The keyed learning methodology we have developed suggests that the constraint we have set—preventing an adversary from predicting classifier behaviour - is enforced. In fact, the adversary would face very strong challenges, because of the significant effect of the key on the results of learning. The key can be in fact of great length and influence all components of the learning process:

example selection, feature selection, learning algorithm choice and corresponding choice of algorithm parameters, timing of learning sessions and timing of defacement testing.

## 7. Materials and Methods

Data and material is available at www.di.unito.it/~fpb/KeyedLearningData/. This includes the core software component, and the data and web links to be used for the experiments in Sections 5.1 and 5.2. The data for the experiments in Section 5.3 are not provided since the owner of the web site being monitored did not provide a corresponding permission.

**Author Contributions:** F.B. developed the adversarial methodology and wrote most of the paper. F.C. (Fabio Cogno) developed the software component containing the *keyed learning* mechanisms. D.R. and F.C. (Fabio Carretto) developed the other system components and carried out the experimentation.

**Funding:** This research received no external funding.

**Acknowledgments:** The authors would like to thank Mario Leone for help with the deployment of the needed IT infrastructure.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

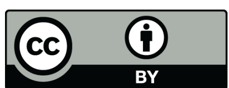
| | |
|---|---|
| WAF | Web Application Firewall |
| IDS/IPS | Intrusion Detection/Prevention System |
| SIEM | Security Incident and Event Management |
| SOC | Security Operations Center |
| CDN | Content Delivery Network |
| CMS | Content Management Systems |

## References

1. Bartoli, A.; Davanzo, G.; Medvet, E. A Framework for Large-Scale Detection of Web Site Defacements. *ACM Trans. Internet Technol.* **2010**, *10*, 10:1–10:37. [CrossRef]
2. Borgolte, K.; Kruegel, C.; Vigna, G. Meerkat: Detecting Website Defacements through Image-based Object Recognition. In Proceedings of the 24th USENIX Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 595–610.
3. Borgolte, K.; Kruegel, C.; Vigna, G. Relevant Change Detection: Framework for the Precise Extraction of Modified and Novel Web-based Content as a Filtering Technique for Analysis Engines. In Proceedings of the 23rd International Conference on World Wide Web Conference (WWW) (IW3C2), Seoul, Korea, 7–11 April 2014.
4. Cooks, A.; Olivier, M. Curtailing web defacement using a read-only strategy. In Proceedings of the 4th Annual Information Security South Africa Conference, Midrand, South Africa, 30 June–2 July 2004.
5. Davanzo, G.; Medvet, E.; Bartoli, A. A Comparative Study of Anomaly Detection Techniques in Web Site Defacement Detection. In *Proceedings of the Ifip Tc 11 23rd International Information Security Conference (SEC 2008)*; Springer: Boston, MA, USA, 2008.
6. Davanzo, G.; Medvet, E.; Bartoli, A. Anomaly detection techniques for a web defacement monitoring service. *Expert Syst. Appl.* **2011**, *38*, 12521–12530. [CrossRef]
7. Enaw, E.E.; Prosper, D.P. A conceptual approach to detect web defacement through Artificial Intelligence. *Int. J. Adv. Comput. Technol.* **2014**, *3*, 77–83.
8. Kanti, T.; Richariya, V.; Richariya, V. Implementation of an efficient web defacement detection technique and spotting exact defacement location using diff algorithm. *Int. Emerg. Technol. Adv. Eng.* **2012**, *2*, 252–256.

9. Liao, X.; Yuan, K.; Wang, X.; Pei, Z.; Yang, H.; Chen, J.; Duan, H.; Du, K.; Alowaisheq, E.; Alrwais, S.; et al. Seeking Nonsense, Looking for Trouble: Efficient Promotional-Infection Detection through Semantic Inconsistency Search. In Proceedings of the IEEE Symposium on Security and Privacy, San Jose, CA, USA, 22–26 May 2016.
10. Verma, R.K.; Sayyad, S. Implementation of Web Defacement Detection Technique. *Int. J. Innov. Eng. Technol.* **2015**, *6*, 134–140.
11. Viswanathan, N.; Mishra, A. Dynamic Monitoring of Website Content and Alerting Defacement Using Trusted Platform Module. In *Emerging Research in Computing, Information, Communication and Applications*; Springer: Singapore, 2016.
12. Kumar, C. 7 Website Defacement Monitoring Tools for Better Security. Last Updated August 2016. Availiable online: https://geekflare.com/website-defacement-monitoring/ (accessed on 27 July 2019).
13. Bartoli, A.; Davanzo, G.; Medvet, E. The Reaction Time to Web Site Defacements. *IEEE Internet Comput.* **2009**, *13*, 52–58. [CrossRef]
14. Chee, W.O. Web Defacements and Data Leakages—Twin Towers website threats. In Proceedings of the RSA Conference, Singapore, 22 July 2016.
15. Zone-H.org. Statistics Report 2008–2016. January 2017. Available online: http://www.zone-h.org/stats/ymd (accessed on 27 July 2019).
16. Zone-H.org. Defacement Portal and Reporting Platform, Active from March 2002 to Present, 2019. Available online: http://www.zone-h.org/ (accessed on 27 July 2019).
17. Maggi, F.; Balduzzi, M.; Flores, R.; Gu, L.; Ciancaglini, V. Investigating Web Defacement Campaigns at Large. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (AsiaCCS 2018), Incheon, Korea, 4 June 2018; pp. 443–456.
18. Biggio, B.; Fumera, G.; Roli, F. Adversarial Pattern Classification Using Multiple Classifiers and Randomization. In Proceedings of the 12th Joint IAPR International Workshop on Structural and Syntactic Pattern Recognition, Orlando, FL, USA, 4 December 2008; pp. 500–509.
19. Chen, Y.; Wang, W.; Zhang, X. Randomizing SVM Against Adversarial Attacks Under Uncertainty. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), Melbourne, Australia, 3–6 June 2018.
20. Bulò, S.R.; Biggio, B.; Pillai, I.; Pelillo, M.; Roli, F. Randomized Prediction Games for Adversarial Machine Learning. *IEEE Trans. Neural Netw. Learn. Syst.* **2017**, *28*, 2466–2478. [CrossRef] [PubMed]
21. Barreno, M.; Nelson, B.; Sears, R.; Joseph, A.D.; Tygar, J.D. Can Machine Learning be Secure? In Proceedings of the 2006 ACM Symposium on Information, Computer and Communi-cations Security (AsiaCCS), Taipei, Taiwan, 21–24 March 2006; pp. 16–25.
22. Lomte, V.; Patil, D. Survey on Keyed IDS and Key Recovery Attacks. *Int. J. Sci. Res.* **2015**, *4*, 12.
23. Mrdovic, R.S.; Drazenovic, B. KIDS—Keyed Intrusion Detection System. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Bonn, Germany, 8–9 July 2010; pp. 173–182.
24. Perdisci, R.; Ariu, D.; Fogla, P.; Giacinto, G.; Lee, W. McPAD: A Multiple Classifier System for Accurate Payload-based Anomaly Detection. *Comput. Netw.* **2009**, *5*, 864–881. [CrossRef]
25. Wang, K.; Parekh, J.; Stolfo, S. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection, Hamburg, Germany, 20–22 September 2006.
26. Bergadano, F.; Carretto, F.; Cogno, F.; Ragno, D. Defacement Response via Keyed Learning. In Proceedings of the 8th IEEE IISA Conference International Conference on Information, Intelligence, Systems & Applications (IISA), Larnaca, Cyprus, 27–30 Augaut 2017.
27. Bergadano, F.; Gunetti, D.; Picardi, C. User Authentication through Keystroke Dynamics. *Acm Trans. Inf. Syst. Secur.* **2002**, *5*, 367–397. [CrossRef]
28. Anwar, S.; Zain, J.M.; Zolkipli, M.F.; Inayat, Z.; Khan, S.; Anthony, B.; Chang, V. From Intrusion Detection to an Intrusion Response System: Fundamentals, Requirements, and Future Directions. *Algorithms* **2017**, *10*, 39. [CrossRef]
29. Munaiah, N.; Meneely, A.; Wilson, R.; Short, B. *Are Intrusion Detection Studies Evaluated Consistently? A Systematic Literature Review*; Technical Report; University of Rochester; Rochester, NY, USA, 2016.

30. Parrend, P.; Navarro, J.; Guigou, F.; Deruyver, A.; Collet, P. Foundations and applications of artificial Intelligence for zero-day and multi-step attack detection. *Eurasip J. Inf. Secur.* **2018**, *2018*, 4. [CrossRef]
31. Kearns, M.; Li, M. Learning in the presence of malicious errors. *Siam Comput.* **1993**, *22*, 807–837. [CrossRef]
32. Huang, L.; Joseph, A.D.; Nelson, B.; Rubinstein, B.; Tygar, J.D. Adversarial Machine Learning. In Proceedings of the ACM Workshop on AI and Security, AISec'11, Chicago, IL, USA, 21 October 2011; pp. 43–57.
33. Lowd, D.; Meek, C. Adversarial Learning. In Proceedings of the 11th ACM SIGKDD international conference on Knowledge Discovery in Data Mining, Chicago, IL, USA, 21–24 August 2005; pp. 641–647.
34. Šrndic, N.; Laskov, P. Practical Evasion of a Learning-Based Classifier: A Case Study. In Proceedings of the IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014.
35. Tapiador, J.E.; Orfila, A.; Ribagorda, A.; Ramos, B. Key-Recovery Attacks on KIDS, a Keyed Anomaly Detection System. *IEEE Trans. Dependable Secur. Comput.* **2015**, *12*, 312–325. [CrossRef]
36. Aggarwal, C.; Pei, J.; Zhang, B. On privacy preservation against adversarial data mining. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 20–23 August 2006; pp. 510–516.
37. Biggio, B.; Corona, I.; Maiorca, D.; Nelson, B.; Šrndic, N.; Laskov, P.; Giacinto, G.; Roli, F. Evasion Attacks against Machine Learning at Test Time. In *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD, Prague, Czech Republic, 23–27 September 2013*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; Volume 8190.
38. Bendale, R.; Gokhale, K.; Nikam, S.; Dhore, A. KIDS: Keyed Anomaly Detection System. *Int. J. Adv. Eng. Res. Dev.* **2017**, *12*, 312–325.
39. Bergadano, F.; Giordana, A. A Knowledge Intensive Approach to Concept Induction. In Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, MI, USA, 12–14 June 1988; pp. 305–317.
40. Arkkom, J.; Carrara, E.; Lindholm, F.; Naslund, M.; Norman, K. *MIKEY: Multimedia Internet KEYing*; IETF RFC 3820; IETF: Fremont, CA,USA, 2004.
41. Chen, L. *Recommendation for Key Derivation Using Pseudorandom Functions*; NIST Special Publication 800-108; NIST: Gaithersburg, MD, USA, 2009.
42. Dierks, T.; Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.2*; IETF RFC 5246; IETF: Fremont, CA,USA, 2008.
43. Xiao, H.; Brown, B.B.G.; Fumera, G.; Eck-ert, C.; Roli, F. Is feature selection secure against training data poisoning? In Proceedings of the 32nd International Conference on Machine Learning (ICML'15), Lille, France, 6–11 July 2015.
44. Hancock, T. On the Difficulty of Finding Small Consistent Decision Trees. Unpublished manuscript. Harvard University: Cambridge, MA, USA, 1989.

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).