2019 IEEE European Symposium on Security and Privacy (EuroS&P)

Mitch: A Machine Learning Approach to the Black-Box Detection of CSRF Vulnerabilities

Stefano Calzavara
Università Ca' Foscari
calzavara@dais.unive.it

Mauro Conti
Università di Padova
conti@math.unipd.it

Riccardo Focardi
Università Ca' Foscari
focardi@dais.unive.it

Alvise Rabitti
Università Ca' Foscari
alvise.rabitti@unive.it

Gabriele Tolomei
Università di Padova
gtolomei@math.unipd.it

***Abstract*—Cross-Site Request Forgery (CSRF) is one of the oldest and simplest attacks on the Web, yet it is still effective on many websites and it can lead to severe consequences, such as economic losses and account takeovers. Unfortunately, tools and techniques proposed so far to identify CSRF vulnerabilities either need manual reviewing by human experts or assume the availability of the source code of the web application.**

In this paper we present Mitch, the first machine learning solution for the black-box detection of CSRF vulnerabilities. At the core of Mitch there is an automated detector of *sensitive* HTTP requests, i.e., requests which require protection against CSRF for security reasons. We trained the detector using supervised learning techniques on a dataset of 5,828 HTTP requests collected on popular websites, which we make available to other security researchers. Our solution outperforms existing detection heuristics proposed in the literature, allowing us to identify 35 new CSRF vulnerabilities on 20 major websites and 3 previously undetected CSRF vulnerabilities on production software already analyzed using a state-of-the-art tool.

I. Introduction

Cross-Site Request Forgery (CSRF) is one of the simplest web attacks to understand, yet it has consistently been one of the top web security threats since its discovery in the early 2000s. In a CSRF attack, a malicious website forces the web browser to perform authenticated, security-sensitive operations at a target web application by means of cross-site requests, without any involvement of the browser's user. This can be done by using just standard HTML tags and JavaScript, making CSRF attempts trivial to perform and forcing security-sensitive web developers to implement solutions to filter out malicious cross-site requests abusing authentication.

Robust defenses against CSRF are well-known [2], but still a significant number of modern web applications was shown to be vulnerable to this class of attacks [28], [34]. The main challenge to face when implementing protection against CSRF is that one has to strike a delicate balance between security and usability. As a matter of fact, the Web is built on top of cross-site requests, most of which are not malicious. Hence, web developers have to choose carefully which operations can only be made available to same-site requests without breaking the website functionality, and implement appropriate security checks on cross-site requests. It is thus easy to accidentally leave room for CSRF attacks, which motivated recent research on automated CSRF detection [28].

Deemon is the first research tool that automatically detects CSRF vulnerabilities [28]. Technically, Deemon is a model-based security testing framework based on a runtime monitor implemented in the PHP interpreter. Although Deemon proved to be very effective on existing open-source web applications, it is a language-dependent analyzer, which only works on PHP applications whose source code is available for dynamic analysis. Overcoming this limitation is of paramount importance to advance the practical impact of the research on the automated detection of CSRF vulnerabilities, because existing web applications are often developed using several different technology stacks, and their source code might not be fully available for analysis in many real-world cases.

Black-box security testing is thus an appealing approach to detect CSRF vulnerabilities, but previous research highlighted that existing web application scanners are not effective in this task. For instance, [3] notes that an existing commercial tool does not report CSRF vulnerabilities "due to the difficulty of determining which forms in the application require protection from CSRF". In fact, correctly detecting *sensitive* HTTP requests is the first major challenge to solve in order to propose an automated black-box detection tool for CSRF vulnerabilities. Unfortunately, previous heuristics from the literature [26], [30] give an unacceptably high number of false positives, as we show in Section IV-E. Thus, security practitioners are often forced to use manual penetration testing tools like Burp[1] and ZAP[2] to detect and test for CSRF vulnerabilities. These tools require users to first *manually identify* the sensitive HTTP requests using their understanding of the web application, and then rely on techniques like those discussed in the OWASP Testing Guide[3] to confirm the attack by running the generated CSRF proof of concept in a web browser to *visually check* its outcome. This is a complex and time-consuming manual task, which we aim to significantly automate and improve.

To achieve this goal, we present Mitch, the first machine learning solution for the black-box detection of CSRF vulnerabilities. At the core of Mitch there is an automated detector of sensitive HTTP requests that outperforms existing detection heuristics proposed in the literature [26], [30].

A. Contributions

More specifically, we contribute to the field of the detection of CSRF vulnerabilities as follows:

[1] https://portswigger.net/vulnerability-scanner

[2] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[3] https://www.owasp.org/index.php/Testing_for_CSRF_(OTG-SESS-005)

DOI 10.1109/EuroSP.2019.00045

1) We develop a methodology to manually identify sensitive HTTP requests on existing web applications with limited effort. We use this approach to build a dataset of 5,828 HTTP requests from 60 popular websites of the Alexa ranking, including 939 sensitive requests. We make the dataset available online, so as to provide a ground truth for future research on the automated detection of sensitive HTTP requests. This manual process is performed only once to train the classifier and is not part of Mitch (Section III).
2) Starting from our dataset, we investigate which features are useful to discriminate between sensitive and insensitive HTTP requests. We then use our dataset to train and test a range of machine-learned classifiers, showing that sensitive HTTP requests can be identified with high accuracy. We experimentally show that our classifiers outperform existing detection heuristics proposed in the literature [26], [30]. These classifiers could be integrated in tools like Burp and ZAP to simplify the task of penetration testers (Section IV).
3) We use our best-performing classifier as a building block for Mitch, the first machine learning solution for the black-box detection of CSRF vulnerabilities. Mitch is a language-agnostic tool, based on a new CSRF detection heuristic, which operates without having access to the source code of the web application to test. This makes it suited to analyze both open- and closed-source web applications, potentially developed using different programming languages (Section V).
4) We experimentally show the effectiveness of Mitch by exposing 35 new CSRF vulnerabilities on 20 major websites and 3 new CSRF vulnerabilities on production software which was already analyzed with Deemon [28]. We also assess that the number of false positives and false negatives returned by Mitch is limited, with just 12 false positives among 47 detected CSRFs and only 7 false negatives overall (Section VI).

To further clarify the scope of our contribution, we anticipate and stress that our classifier is intended to be integrated in penetration testing tools for finding CSRF vulnerabilities, *not* to be used for CSRF attack detection. For this reason, it is not designed to be resilient to adversarial manipulations [1]. The study of techniques to deal with this intriguing yet orthogonal aspect is left to future work.

II. Background

In this section, we provide an overview of the basics of cross-site request forgery and supervised learning.

A. Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is a well-known web attack that forces a user into submitting unwanted, attacker-controlled requests towards a vulnerable web application in which she is currently authenticated. The key concept of CSRF is that the malicious requests are routed to the web application through the user's browser, hence they might be indistinguishable from intended benign requests which were actually authorized by the user.

1) Attack Description: A typical CSRF attack works as follows:

1) Alice logs into an honest yet vulnerable web application. Session authentication is implemented through a session cookie that is automatically attached by the browser to any subsequent request towards the web application.
2) Using the same browser, Alice visits Eve's malicious website, which sends a *cross-site* request to the vulnerable web application using HTML or JavaScript.
3) Since the request includes Alice's cookies, it is processed by the vulnerable web application in the authentication context of Alice. This way, Eve can trigger security-sensitive actions on Alice's behalf.

It is worth noticing that CSRF is purely a *web attack*, since it does not require the attacker to intercept or modify user's requests and responses: it is enough that the user visits the attacker's malicious website, from which the attack is launched. Thus, web applications which suffer from CSRF vulnerabilities are potentially exploitable by any malicious website on the Web.

2) Current Fixes and Mitigations: To prevent CSRF attacks, web developers have to implement explicit protection mechanisms [2]. If adding extra user interaction does not affect usability too much, it is possible to force re-authentication or use one-time passwords or CAPTCHAs [36] to prevent cross-site requests going through unnoticed. In many cases, however, automated prevention is preferred and cross-site requests are filtered out by using any of the following techniques:

- checking the value of standard HTTP request headers such as `Referrer` and `Origin`, indicating the page originating the request;
- checking the presence of custom HTTP request headers like `X-Requested-With`, which can only be set on same-origin AJAX requests;
- checking the presence of unpredictable anti-CSRF tokens, set by the server into sensitive forms.

Each of these mechanisms has a few subtleties that make it unreliable in some cases, and OWASP recommends to always implement anti-CSRF tokens together with appropriate header checks, suggesting that a single mechanism might not suffice[4]. Alternatively, web developers may use SameSite cookies, a relatively new browser-based defense mechanism which allows one to mark cookies which must not be attached to cross-site requests, thus preventing CSRF attacks in the browser.

3) Login CSRF: There exists a variation of CSRF known as *login CSRF* [2], where the attacker Eve forces her victim Alice into logging into Eve's account at a vulnerable web application. This can be used for instance to steal confidential information that will be sent to Eve's account instead of Alice's account. We will not consider protection against login

[4] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

CSRF in this paper; we refer the interested reader to [34] for some recent results on the topic.

B. The Supervised Learning Problem

One of the most popular scenarios where machine learning has proved useful is *supervised learning*. This is the task of learning a function that maps an input to an output based on a sample of observed input-output pairs (*instances*), which is usually referred to as *training set*.

More formally, let $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1 \ldots m}$ be a training set. Each $\mathbf{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ is an n-dimensional vector of *features* representing the i-th input and $y_i \in \mathcal{Y}$ is its corresponding output value. In this paper, we focus on *binary classification* problems where $\mathcal{Y} = \{0, 1\}$ and each y_i is a bit known as the *class label*. Supervised learning assumes the existence of an unknown *target* function $f : \mathcal{X} \mapsto \mathcal{Y}$ that maps any feature vector to its corresponding output. The goal is therefore learning the function $\hat{f}$ that best approximates f on $\mathcal{D}$ from a set of *hypotheses* $\mathcal{H}$, using a *loss* function ℓ to estimate the cost of the prediction errors. Specifically, learning $\hat{f}$ reduces to the following optimization problem:

$$\hat{f} = \operatorname*{argmin}_{f^* \in \mathcal{H}} \ell(f^*, \mathcal{D}).$$

Different estimates $\hat{f}$ can be learned depending on the choice of $\mathcal{H}$ and ℓ. Covering these aspects more in detail is outside the scope of this work; we however refer the interested reader to [14] for an in-depth discussion.

III. Dataset Construction

In this section, we first give a practical definition of sensitive request and we describe how we collect and label a real-world dataset of sensitive and insensitve requests, taking also into account possible ethical implications.

A. Sensitive Requests

Before building our dataset, we first need a rigorous definition of sensitive request which drives the labeling process. We adapt this definition from existing work on CSRF detection, which defined a notion of CSRF vulnerability [28].

Definition 1 (Sensitive Request)**.** *An HTTP request is* sensitive *if and only if:* (i) *it causes a security-relevant state change in the web application which processes it; and* (ii) *it is processed within a valid authentication context of a registered user.*

Identifying sensitive requests is a prerequisite to the detection of CSRF vulnerabilities, because the latter can only be exploitable when the former can be crafted by the attacker, i.e., when the attacker knows all the required parameters and values of the requests.

Unfortunately, detecting sensitive requests is generally hard, because it requires both to understand the semantics of the web application and to observe the presence of server-side state changes. While the first requirement can be reliably fulfilled by human experts, the second one may be unfeasible even for them, in particular when the web application code is unavailable or too complex to be analyzed. For this reason, it is worth introducing the class of the *visibly* sensitive requests.

Definition 2 (Visibly Sensitive Requests)**.** *A sensitive request is* visibly sensitive *if and only if its security-relevant web application state change is visible at the browser side upon processing the corresponding response.*

Visibly sensitive requests are routinely used to implement a wide range of functionalities which need to be protected against CSRF. Notable examples include:

- social media actions, e.g., liking and disliking contents, social sharing between websites, posting on personal profile pages and timelines;
- profile management, e.g., changing user preferences, setting a birth date, uploading a profile picture;
- e-payments and (more generally) online transactions, which most often implement visual indicators to keep track of the transaction state, e.g., the number of items currently stored in the shopping cart.

Examples of sensitive requests which are not visibly sensitive include all the tracking and profiling operations often implemented by modern websites, whose corresponding state changes amount to the update of server-side logs, e.g., for targeted advertising. Though these requests should certainly be protected against CSRF to safeguard the user experience, they cannot be detected as sensitive without having direct access to the web application code. Moreover, they are way less straightforward to exploit for an attacker, who has to infer the inner workings of the tracking system to abuse it. From now on, we only focus on visibly sensitive requests and we just refer to them as sensitive for simplicity.

B. Request Collection and Labeling

We developed a browser extension (HTTP-Tracker) to manually label HTTP requests sent from a web application as sensitive or insensitive. The main challenge in the extension development was the definition of heuristics which make the manual labeling feasible in practice, given the size and complexity of modern web applications and the overwhelming amount of generated HTTP requests.

1) Collecting requests: HTTP-Tracker uses the webRequest API of the extension system to monitor all the requests of type `main_frame`, `sub_frame` and `xmlhttprequest` sent by the browser. This restriction effectively filters out a huge number of requests, e.g., for image and script inclusion, which are very likely to be insensitive. All the monitored requests whose URL matches a configurable pattern are then logged for manual labeling. We use such pattern to focus on a specific website in the labeling process: for instance, when navigating `www.example.com`, we only log requests sent to `example.com` and its sub-domains. This means that we are only interested in state changes affecting the same web application that we are navigating, which we assume to be possibly hosted on multiple related domains.

For each logged request, the extension separately stores its method, its URL, and its parameters, along with their corresponding values. Requests which are identical up to the values of their parameters are only logged once to simplify the labeling process. To correctly process the parameters, the extension distinguishes between GET and POST requests. GET parameters are parsed based on the standard query string format, while POST parameters are extracted from the body of the request using a set of existing parsers for the most common content types used in the wild, e.g., `multipart/form-data` and `application/json`; the right parser is chosen by inspecting the value of the `Content-Type` header of the request. By clicking on the icon of HTTP-Tracker, a human expert can access the list of the logged requests and proceed to mark them as sensitive or insensitive.

2) Labeling methodology: To perform the labeling of the requests to a web application w, we proceed as follows. We first access the homepage of w from a fresh browser and we mark all the outgoing requests as insensitive, because no security-relevant website functionality was yet exercised. We then authenticate at w and we use our understanding of the web application semantics to visually identify how to exercise the functionalities which are likely to trigger sensitive requests, e.g., updating profile information or liking a content. After each click, we let the browser complete page loading and we proceed to manually label the logged HTTP requests.

Perhaps surprisingly, despite the complexity of modern web applications, this is a trivial task in the large majority of cases. Though browsers send a huge number of requests upon page rendering, the filters implemented in HTTP-Tracker are often strict enough to ensure that only one request per click needs to be labeled, hence the right label is a direct consequence of the clicked page element. In the few cases where multiple requests need to be labeled after a single click, we rely on a manual inspection of the logged requests to understand their semantics, possibly using the developer tools available in the web browser to deal with the subtlest cases. For instance, we note that replaying requests is a convenient strategy to collect more information about their sensitivity, e.g., the server might answer that the requested operation was already performed, which suggests that a server-side state change happened after the original request.

C. A Dataset of Labeled Requests

To build our dataset, we accessed the Alexa ranking and we selected a total of 60 websites featuring authenticated access from the top sites of each of the available categories. This way, we were able to cover web applications with fundamentally different scopes, such as social networks, e-shops, and adult websites. This diversification is pivotal in ensuring that the dataset of HTTP requests collected is representative enough to be unbiased. We then created personal accounts at the 60 websites and we performed the manual labeling of their HTTP requests using HTTP-Tracker, as described in the previous section. We do not claim nor target full coverage of all the possible website functionalities, both for technical and ethical reasons, but we tried to be as much comprehensive as possible in our navigation experience. The labeling was performed by two of the authors, who conducted a systematic mutual verification of their results. In the few cases where there was no agreement between the two experts, requests have not been included in the dataset to avoid the introduction of inaccurate information.

In the end, we were able to collect 6,312 HTTP labeled requests. From this original dataset, we filtered out any HTTP request whose method was different from GET and POST. The rationale of this last choice is that PUT and DELETE requests turned out to be all sensitive, while OPTIONS requests were all insensitive, which suggests that the classification problem for these cases is trivial and does not really require the use of machine learning techniques. Overall, we reduced the size of the dataset by around 7.7% to 5,828 HTTP requests, including 939 sensitive requests[5].

D. Ethical Implications

Sensitive requests cause by definition security-relevant state changes in the web application which processes them, hence they should not be forged lightheartedly. We used our human understanding of the analyzed web applications to guarantee that the sensitive requests we sent to construct our dataset did not have any negative impact on others, be they websites users, web developers, server administrators, or website owners.

A few relevant ethical considerations follow. The first observation we make is that a significant fraction of the sensitive requests we sent only affected the state of our own sessions: for instance, this is the case for requests which updated our profile information or personal preferences. When a sensitive request could affect other people, e.g., a friend request was sent, we always ensured that it was *reversible*, and we proceeded to restore the web application state to its original form just after labeling the request. At the end of our experiments, we deleted the personal accounts we created on all the websites which provided such functionality. Not only this ensured that all the actions performed by the account were undone, but also allowed us to include in the dataset additional sensitive requests used for account deletion.

IV. HTTP Request Classification

Having a dataset of HTTP requests labeled as sensitive and insensitive allows us to frame the problem of identifying requests that need to be protected against CSRF into a *binary classification* problem, as defined in Section II-B.

A. Feature Engineering

The first effort when designing a machine learning pipeline usually consists in finding the most appropriate features to represent the objects of the domain of interest: this process is called *feature engineering*. This task often requires a significant amount of human effort, backed up by a strong domain knowledge. In exchange for that, this approach is able to build predictive models operating on *human-understandable* characteristics of objects in the domain. Indeed, we are not only interested in correctly identifying sensitive HTTP requests, but

[5] We make the dataset available at https://github.com/alviser/mitch.

also in understanding which (interpretable) features have led to such predictions. For this reason, more recent techniques that are able to automatically extract representations from data based on deep learning are not a viable option for us, since those learned features are typically hard to attribute to interpretable properties [23]. Although models resulting from training (deep) neural networks have proven very effective in specific perceptual tasks, such as computer vision and natural language processing [22], [4], trading off a highly accurate predictive model with a possibly lower performing one which operates on human-understandable features is particularly appreciated in some settings like ours [35].

In this work, we consider the feature space $\mathcal{X}$ being made of 49 dimensions, each one capturing a specific property of HTTP requests. Those can be logically organized into 3 categories: *Structural*, *Textual*, and *Functional*.

1) Structural: This category of features describes structural properties of an HTTP request. More precisely, we define the following set of numerical features:

- `numOfParams`: the total number of parameters of the request;
- `numOfBools`: the number of request parameters bound to a boolean value;
- `numOfIds`: the number of request parameters bound to an *identifier*, i.e., a hexadecimal string, whose usage was empirically observed to be common in our dataset;
- `numOfBlobs`: the number of request parameters bound to a *blob*, i.e., any string which is not an identifier;
- `reqLen`: the total number of characters in the request, including parameter names and values.

While one might devise more sophisticated techniques to type request parameters, HTTP requests have a very weak structure and it is hard to come up with general yet accurate typing techniques which could be used without interacting with the server multiple times [37]. Avoiding this, in turn, is important for the online detection of sensitive requests.

2) Textual: This category of features captures textual characteristics of HTTP requests and is based on a small manually-curated vocabulary of keywords V that may occur in the request, resulting from a manual inspection of sensitive requests from a sample of real-world websites considered in our dataset. More specifically, we only consider binary features of the following forms:

- $word$`InPath`, where $word \in V$ means the presence of the string $word$ in the request path;
- $word$`InParams`, where $word \in V$ means the presence of the string $word$ in any parameter name of the request.

The vocabulary V includes the following 21 keywords, which have been selected as possible signals of sensitive requests, according to common sense and a preliminary inspection of the part of our dataset which is reserved for training: *create*, *add*, *set*, *delete*, *update*, *remove*, *friend*, *setting*, *password*, *token*, *change*, *action*, *pay*, *login*, *logout*, *post*, *comment*, *follow*, *subscribe*, *sign*, and *view*.

3) Functional: This category of features indicates the HTTP method associated to the request. We consider just the following two binary features:

- `isGET`: the HTTP request method is GET;
- `isPOST`: the HTTP request method is POST.

There are no additional alternatives, because our dataset only includes GET and POST requests.

B. Data Exploration

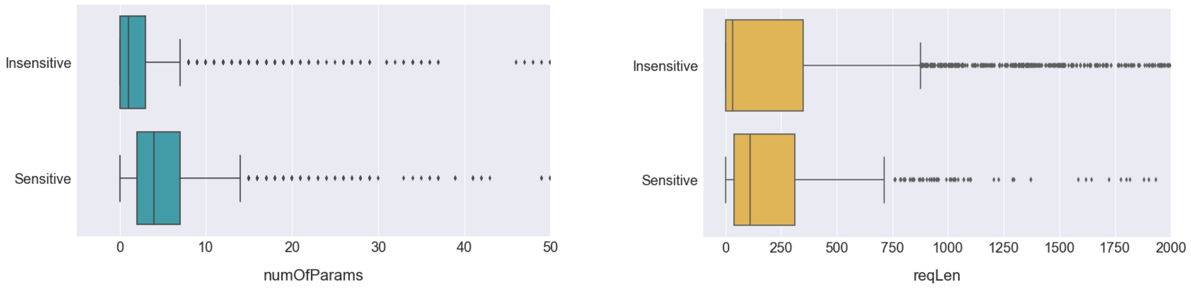
Before diving into the core stage of the machine learning pipeline, in this section we describe an exploratory analysis of our labeled dataset of HTTP requests, represented using the set of features discussed above.

1) Structural: We start by investigating two representative numerical features: `numOfParams` and `reqLen`. The former counts the number of request parameters, while the latter measures the request length, as specified above. Figure 1 depicts two box plots, one for each of the aforementioned features, showing how those are distributed across the two class labels. The average number of parameters of sensitive requests is around 6.27, while for insensitive ones this is 3.43. From Figure 1(a), it is evident that the median value (i.e., the vertical bar inside the colored rectangle) of `numOfParams` is significantly higher for sensitive than for insensitive HTTP requests (i.e., 4 vs. 1). More generally, the inter-quartile range of the values of this feature (i.e., the width of the rectangle) is larger, therefore suggesting that sensitive HTTP requests are characterized by a higher number of parameters. This is still true even considering the presence of extreme values, also called *outliers*, on both sensitive and insensitive requests, which are represented as scattered dots outside the range delimited by the two vertical bars in the plot.

Apparently, sensitive requests tend to be shorter than insensitive requests on average (322 vs. 592 characters). However, this is due to the presence of outliers, as shown in Figure 1(b), which shift the average of insensitive requests towards higher figures. In fact, the median request length – which is more robust to the presence of outliers – is significantly larger for sensitive requests (111 vs. 34 characters). This suggests that the length of sensitive requests is somewhat more consistent and bounded within a smaller range of generally higher values.

2) Textual: Figure 2 shows 20 out of the 42 features in this category, and their ability in detecting sensitive HTTP requests independently from the others. More specifically, each individual plot refers to a textual (binary) feature; for each of the two possible values this feature can take, a histogram shows the ratio between sensitive and insensitive requests having that specific value of the feature. For example, in the top-left plot around 84.9% of the total number of requests having no “create” keyword in their path are insensitive, and the remaining 15.1% are sensitive. On the same plot, the other histogram shows that about 38.6% of the total requests having “create” keyword in their path are insensitive, and the remaining 61.4% are sensitive.

It is worth noticing that features that do not seem highly discriminant if examined alone may turn into good predic-



(a) Distribution of `numOfParams`

(b) Distribution of `reqLen`

Fig. 1: Distribution of two representative structural features across class labels

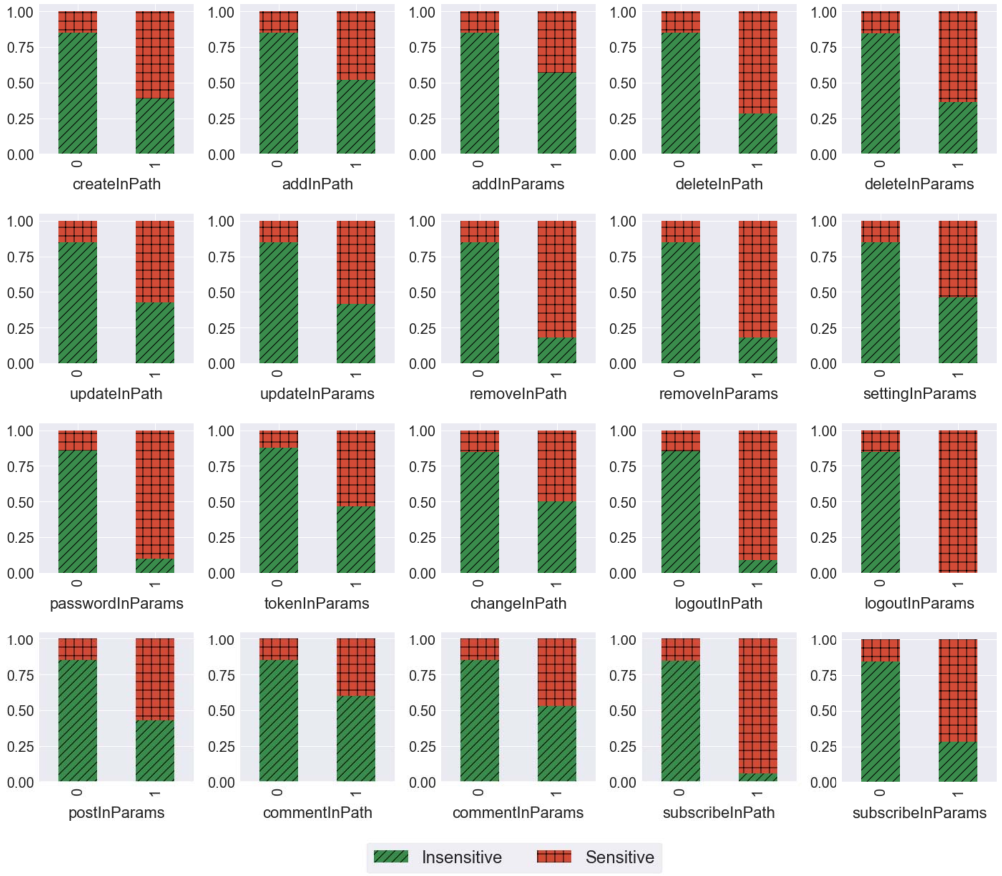


Fig. 2: Distribution of class labels across the top-20 most predictive textual features

tors when considered together with other features. Choosing which features should be used for training a classifier – also known as *feature selection* – is one of the core tasks of machine learning [18], [25], and will be discussed later. Still, a few preliminary observations can be made from the plots in Figure 2. For example, most of the times a request contains a term related to irreversible actions, such as “delete” or “remove”, it is labeled as sensitive. Similarly, requests containing authentication-related terms, such as “password”, are also sensitive most of the times.

3) Functional: Finally, Figure 3 shows the distribution of the two features `isGET` and `isPOST` across sensitive and insensitive requests. We observe that about 30% of POST requests are labeled as sensitive, while only 5% of GET requests are sensitive. This is compliant to what one would expect, because the GET method is intended to denote operations

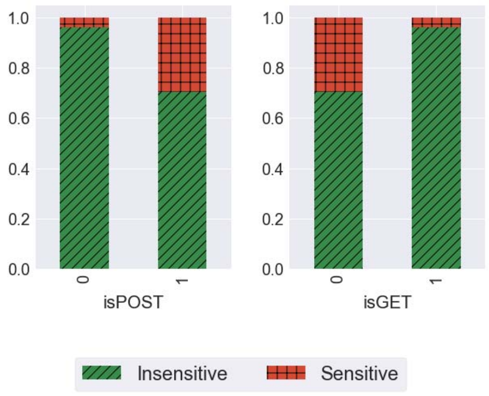


Fig. 3: Distribution of class labels across the two functional features

which are both idempotent and safe. Note that the plots are symmetric since our dataset contains only GET and POST requests, i.e., if `isGET` = 0 then `isPOST` = 1, and vice versa.

C. Learning Binary Classifiers

We consider the following set of hypotheses to pick our classifier $\hat{f}$ from: Logistic Regression (LogReg) [38], Support Vector Machines (SVM) [12], Decision Trees (DT) [6], Random Forests (RF) [20], [5], and Gradient Boosted Decision Trees (GBDT) [17].

Since no major data preprocessing is needed, we start by splitting our original dataset $\mathcal{D}$ into two distinct portions: $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$. The former accounts for 90% of the total instances in $\mathcal{D}$, whereas the latter contains the remaining 10%. Due to severe class imbalance between sensitive and insensitive requests (16% vs. 84%, respectively), we use *stratified* random sampling to perform the splitting [27]. This way, we guarantee that instances in both $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ follow the same class distribution of the one observed on the whole dataset $\mathcal{D}$, which reduces the selection bias.

To choose between different hypotheses, as well as different sets of features and hyperparameters[6] within each individual hypothesis, we rely on a standard technique known as *nested cross-validation* [11] performed on $\mathcal{D}_{\text{train}}$, in combination with a common evaluation metric, which is robust against class imbalance, i.e., ROC AUC [16]. Once the best-performing model is chosen using the methodology above, we evaluate it on the independent, held-out, test set $\mathcal{D}_{\text{test}}$. Eventually, the final classifier $\hat{f}$ is obtained by re-training the best model on the whole original dataset $\mathcal{D}$.

D. Offline Evaluation

We conducted our experiments using the implementations of the hypotheses we focused on (i.e., LogReg, SVM, DT, RF, and GBDT) as provided by the Python `scikit-learn`[7] package for machine learning.

LogReg	SVM	DT	RF	GBDT
0.907 (± 0.03)	0.906 (± 0.03)	0.882 (± 0.02)	**0.932** (± 0.02)	0.930 (± 0.02)

TABLE I: Avg. (± std. dev.) ROC AUC scores after nested cross-validation on $\mathcal{D}_{\text{train}}$

In Table I, we report the performance obtained by each hypothesis in response to a 10-fold nested cross-validation run on $\mathcal{D}_{\text{train}}$. The best-performing model is RF, which reaches the maximum value of ROC AUC at 0.932.

We choose RF and test three different values of the hyperparameter controlling the number of trees of the forest: 100, 500, and 1,000. From our experiments, the best value of this hyperparameter is 500. At the same time, we select the subset of most likely relevant features. To do so, we extract the subset of features $\mathcal{X}^{(i)} \subseteq \mathcal{X}$ of size i with the highest chi-squared test statistic against the class label. We test different values $i \in \{5, 15, 25, 35, 45, 49\}$, where 49 corresponds to the number of initial features, and the best classification performance is obtained with $\mathcal{X}^{(45)}$. The four discarded features are: `changeInParams`, `passwordInPath`, `payInPath`, and `viewInParams`. In Section IV-F, we elaborate more on this and how to rank features according to their importance.

To further validate the quality of our learned RF classifier trained on the entire $\mathcal{D}_{\text{train}}$ (using the set of features and hyperparameters selected with the method above), we measure its performance on $\mathcal{D}_{\text{test}}$, i.e., the portion of dataset initially held-out for testing. Accounting for 10% of the whole dataset, $\mathcal{D}_{\text{test}}$ contains 583 instances. The ROC AUC score obtained by our RF classifier on this test set is 0.924, a value which is perfectly compliant with the estimate computed using nested cross-validation (0.932).

In addition to the ROC AUC score, we also report three other standard performance metrics: *precision*, *recall* and F_1, which are defined on top of the number of true positives (*tp*), true negatives (*tn*), false positives (*fp*) and false negatives (*fn*) produced by the classifier. Precision measures the fraction of truly sensitive requests out of all the requests that are labeled as sensitive by the classifier. Instead, recall measures the fraction of truly sensitive requests that the classifier is actually able to predict:

$$\begin{aligned} \textit{precision} &= \frac{tp}{tp+fp}. \\ \textit{recall} &= \frac{tp}{tp+fn}. \end{aligned}$$

One way to aggregate the effect of both measures into a single score is to compute F_1, which is the harmonic mean of precision and recall and is defined as follows:

$$F_1 = 2 \cdot \frac{\textit{precision} \cdot \textit{recall}}{\textit{precision} + \textit{recall}}.$$

[6] A hyperparameter is a parameter whose value is set independently of the learning process, e.g., the number of trees of Random Forest. By contrast, the values of the parameters defining the hypothesis result directly from the training step.

[7] http://scikit-learn.org/

Overall, our best-performing RF classifier achieves the following results in terms of the three metrics discussed above: *precision* = 0.78, *recall* = 0.67, and F_1 = 0.72. We can thus observe it slightly favors accurate predictions (i.e., almost 80% of the requests it labels as sensitive are indeed sensitive), although this comes at the cost of possibly missing a few sensitive instances. Other classifiers might exhibit higher recall scores, i.e., higher coverage of sensitive requests, but the alternative models we tested had all lower values of F_1.

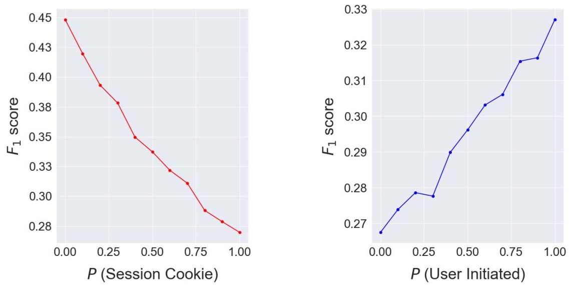
E. Comparison with Baselines

We provide a comparison between the performances of our machine-learned classifier and those obtained by two baselines proposed in the literature [26], [30]. Both of them use heuristics based on common sense arguments to detect sensitive HTTP requests, but were not actually tested against a ground truth.

The first baseline was proposed by Mao *et al.* for the tool BEAP [26]. The authors propose the recipe summarized in Table II(a) for identifying sensitive requests, which exploits four features: the request method, the protocol, the presence of cookies (session or permanent) and the presence of the `Authorization` header. The second baseline was introduced by De Ryck *et al.* for the tool CsFire [30]. The heuristic is described in Table II(b) and is based on three features: the request method, the presence of request parameters and the fact was the request was *user-initiated*. For example, the authors indicate a click on a link as an example of a user-initiated request. However, notice that links can be clicked by JavaScript as well, so it is unclear how user-initiated requests can be reliably identified. We discuss below how we addressed this issue in our comparison.

To provide a fair comparison between our machine-learned classifier and the two heuristics above we proceed as follows. First of all, we use $\mathcal{D}_{\text{test}}$ as our testbed. Then, we re-implement the proposed heuristics and we compute the same metrics used to assess our classifier, i.e., precision, recall, and F_1.

We start by discussing the comparison with BEAP. Though its heuristic is straightforward to implement, there is a complication due to the distinction between session cookies and permanent cookies, which we did not track in our dataset. To overcome this issue, we use a probabilistic approach: whenever we find a GET request in $\mathcal{D}_{\text{test}}$, we simulate the presence of session cookies with probability p and that of permanent cookies with probability $(1 - p)$. We test the performances achieved by the heuristics for different values of p, i.e., $p \in \{0.0, 0.1, \ldots, 0.9, 1.0\}$, and we pick the value which leads to the highest F_1 score. In our experiments, the heuristics reaches its highest performances when $p = 0.0$, i.e., when all the GET requests are marked as insensitive. As to CsFire, we apply the very same approach to simulate whether an HTTP GET request without parameters was user-initiated or not. The heuristic reaches its highest performance when $p = 1.0$, i.e., when all the GET requests with no parameters are marked as insensitive. Figure 4(a) and 4(b) show how the F_1 scores of BEAP and CsFire change when varying the value of p.



(a) BEAP (b) CsFire

Fig. 4: F_1 scores when varying the probability p

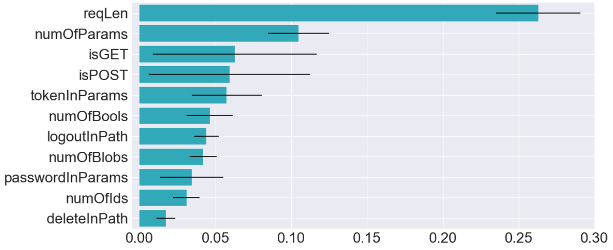


Fig. 5: Top-10 most important features derived from our RF classifier using Gini impurity

Table III reports the computed validity measures of the approaches we tested. We observe that the heuristics implemented in BEAP and CsFire reach a very high recall, but their precision is extremely low: too many insensitive requests are marked as sensitive by these solutions, which produce an unacceptably high number of false positives for practical use cases. Indeed, previous research identified major usability issues in these defense mechanisms due their high misclassification rate [13]. Our classifier, in turn, strikes a much better balance between precision and recall, as testified by its remarkably higher F_1 score.

F. Feature Importance

As a by-product of our learned RF classifier, we can also derive a ranking of feature "importance", i.e., a list of the most predictive features. There are several ways to compute feature importance; in this work, we use the importance score as provided by the `scikit-learn` package, which in turn implements it using *Gini impurity* [6]. Figure 5 shows the top-10 most important features derived from our RF classifier.

We can observe that the two most important features are `reqLen` and `numOfParams`, followed by the two features encoding the request method. This indicates that structural and functional features alone are already good predictors of sensitive requests. Still, few textual features are highly ranked as well, somehow suggesting that natural language signals can also be useful, although their effectiveness strongly depends on the manually-chosen vocabulary of keywords. To overcome this limitation, as future work we plan to investigate more

	GET		POST
	HTTP	HTTPS	
`sess_cookies`	*insensitive*	*sensitive*	*sensitive*
`perm_cookies`		*insensitive*	
`Authorization`	*sensitive*		

(a) BEAP

	GET		POST
	`params`	`no_params`	
`user_init`	*sensitive*	*insensitive*	*sensitive*
`not_user_init`		*sensitive*	

(b) CsFire

TABLE II: Schemas of the heuristics used by BEAP [26] and CsFire [30] to identify sensitive requests

Classifier	*Precision*	*Recall*	F_1
BEAP	0.30	0.89	0.45
CsFire	0.20	**0.97**	0.33
RF	**0.78**	0.67	**0.72**

TABLE III: Validity measures for the tested classifiers

advanced NLP techniques to build our reference vocabulary of terms from an independent dataset of HTTP requests.

V. Mitch: Design and Implementation

In the previous section, we showed that machine-learned classifiers can be used to accurately identify sensitive HTTP requests, hence they can be leveraged as a useful building block for penetration testing tools like Burp and ZAP, in particular to assist users in finding where CSRF proof of concepts should be generated and manually tested.

We now make a further step forward by presenting Mitch, the first tool for the automated black-box detection of CSRF vulnerabilities. Mitch is a browser extension designed on top of our best-performing classifier, trained on the full dataset we collected. The key idea of Mitch is simple and reminiscent of traditional manual techniques for CSRF detection like those advocated in the OWASP Testing Guide[8], but the details of its design are tricky and important for the effectiveness of its automated approach, which we experimentally assess on existing web applications.[9]

A. Key Idea and Challenges

Mitch assumes the possession of two test accounts (say, Alice and Bob) at the website where the security testing is to be performed. This is used to simulate a scenario where the attacker (Alice) inspects sensitive HTTP requests in her session to force the forgery of such requests in the browser of the victim (Bob). Having two test accounts is crucial for the precision of the tool because if the forged requests contain something which is bound to Alice's session, then CSRF against Bob may not be possible. For example, if a website defends against CSRF through the use of an unpredictable user identifier, then Alice's requests will be rejected in Bob's session. The use of two test accounts for CSRF detection has already been advocated in previous work [34] and is part of traditional manual testing strategies.[10]

After installing Mitch in her browser, the security tester first navigates the website as Alice: for every HTTP request detected as sensitive, Mitch stores the content of the corresponding HTTP response. After completing the navigation, Mitch uses the collected sensitive HTTP requests to generate new HTML elements in the extension origin which allow for replaying them. The security tester then authenticates to the website as Bob and Mitch exploits the generated HTML to automatically replay the detected sensitive requests from a cross-site position, which simulates a CSRF attack. Finally, the responses collected for Alice and Bob are compared: if a response received by Bob "matches" the one received by Alice, it means that Alice was able to forge a valid request for Bob's session, hence the attack is considered successful and Mitch reports a potential CSRF vulnerability.

This simple idea brings up a number of challenges for the automation process:

(C1) *Changes in HTTP responses*: Defining a suitable notion of matching HTTP responses for Alice's and Bob's sessions is generally hard, because HTTP responses may include dynamically generated elements, which might realistically differ even when the same idempotent operation is performed multiple times.

(C2) *Changes in session state*: Since the state of Alice and Bob at the website might be different, matching the response received by Bob against the response received by Alice might be an improper way to detect a CSRF vulnerability. For instance, Bob might not be able to perform a sensitive operation because it does not have access to the file `foo`, yet a CSRF attack would work if it targeted the file `bar`.

(C3) *Classification errors*: Even a very accurate classifier might incorrectly mark an insensitive request as sensitive. In this case, there is no CSRF vulnerability and the presence of matching responses for Alice's and Bob's sessions should not raise an alarm.

Clearly, there is no completely accurate way to deal with all these issues, yet one can find useful heuristics which successfully work in many practical cases, as we show in our experimental evaluation.

B. CSRF Detection Algorithm

We start by discussing how we tackled the three challenges presented above to minimize the number of false positives and false negatives in Mitch:

(C1) *Changes in HTTP responses*: Rather than relying on complex techniques to detect matching HTTP responses,

[8] https://www.owasp.org/index.php/Testing_for_CSRF_(OTG-SESS-005)

[9] We make Mitch available online at https://github.com/alviser/mitch

[10] https://support.portswigger.net/customer/portal/articles/1965674-using-burp-to-test-for-cross-site-request-forgery-csrf-

Mitch builds on a notion of *dissimilar* HTTP responses. In general, the dissimilarity of HTTP responses is much easier to check than their similarity, e.g., due to the use of different status codes or content types to denote failures (for example, status codes 401 and 403 are typical ways to denote unauthorized access). When Bob's response is dissimilar from Alice's response, it is likely that Alice's request failed in Bob's session, which might indicate the use of a CSRF protection mechanism.

(C2) *Changes in session state*: When comparing the response received by Bob against the one received by Alice, Mitch does not immediately consider their dissimilarity as a definite evidence that the request of Bob had a different outcome than the one of Alice due to the use of a CSRF protection mechanism. Rather, since different outcomes might come from a difference in the state of Alice's and Bob's sessions, Mitch also replays the original request of Alice in a fresh Alice's session: if the new response received by Alice is dissimilar to the original one, it is likely that session-dependent information is required to process the request, which might indicate the adoption of an anti-CSRF token.

(C3) *Classification errors*: To detect potential false positives produced by the classifier, Mitch replays the original request of Alice without first authenticating to the website, i.e., outside any session: if the received response is dissimilar from the original one, then there is additional evidence that the requested operation required an authenticated context to be performed, which confirms that there exists potential room for CSRF.

More formally, Mitch is based on the notion of *traces*. A trace T_i is a set of pairs (r, s), where r is an HTTP request and s is the corresponding HTTP response, with the constraint that all the requests in T_i are sent in the same authentication context i. We only consider three possible authentication contexts: a for Alice, b for Bob and u for unauthenticated. We also assume that all the requests occurring inside a trace are pairwise distinct and we write $T_i[r] = s$ when $(r, s) \in T_i$ to improve readability.

The vulnerability finding algorithm implemented in Mitch is shown in Algorithm 1. Given a set of sensitive requests $Reqs$, the algorithm returns a set of candidate vulnerabilities $Cand$ by processing the traces T_a, T'_a, T_b and T_u as mentioned in the previous informal overview. Specifically, the algorithm works by first building a set of candidate vulnerabilities, based on sensitive requests which produce dissimilar responses in Alice's session and in the unauthenticated context, which confirms that an authenticated context is required (C3); candidates are then discarded if request replay attempts fail in both Bob's session (C1) and a fresh Alice's session (C2), i.e., the corresponding responses are dissimilar from the one originally received by Alice, which suggests the adoption of anti-CSRF defenses. The algorithm is parametric with respect to the definition of response dissimilarity $\not\sim$, whose current implementation is discussed in the next subsection.

Algorithm 1 CSRF Detection Algorithm

```
procedure FindCSRF($Reqs, T_a, T'_a, T_b, T_u,$)
    $Cand \leftarrow \emptyset$
    for $r \in Reqs$ do
        if $T_a[r] \not\sim T_u[r]$ then                ▷ See point (C3)
            $Cand \leftarrow Cand \cup \{r\}$
    for $r \in Cand$ do
        if $T_a[r] \not\sim T_b[r]$ then                ▷ See point (C1)
            if $T_a[r] \not\sim T'_a[r]$ then           ▷ See point (C2)
                $Cand \leftarrow Cand \setminus \{r\}$
    return $Cand$
```

C. Implementation

Figure 6 summarizes the architecture of Mitch, which is implemented in JavaScript as a browser extension. We employed `sklearn-porter`[11] to export the trained RF classifier as a JavaScript function. The current implementation of Mitch is only compatible with Mozilla Firefox, because it is the only browser whose extension system provides native facilities to access the body of HTTP responses as of now.

Mitch currently builds on the following definition of response dissimilarity $\not\sim$.

Definition 3 (Dissimilar Responses)**.** *Two HTTP responses s and s' are* dissimilar*, written $s \not\sim s'$, if and only if they satisfy any of the following conditions:*

1) *their status code is different;*
2) *their content type is different;*
3) *their content type is* `text/html`*, but the length of their payloads differs of more than 1%;*
4) *their content type is* `text/json`*, but their payloads validate against different JSON schemas;*
5) *their content type is neither* `text/html` *nor* `text/json`*, but their payloads are different.*

We observed that this definition works well on existing web applications, since we noticed that the first, the second, and the fourth clauses are very popular patterns to discriminate between success and failure in the wild. Yet, we do not exclude that further improvements are possible: in particular, we notice that the empirical threshold used in the third clause might need to be adjusted after a more extensive experimental analysis. The threshold we chose is motivated by the observation that the difference between the HTML sent to authenticated users and the HTML sent in unauthenticated contexts is typically high, e.g., because a private area is shown in the former case, while a login page is shown in the latter case. Instead, the difference in the HTML sent to two different authenticated user is much lower on average. By keeping a small threshold in the third clause, it is likely that a request which requires authentication is initially added to the set of candidates and is only kept there if there is strong evidence that the replayed requests succeeded. Hence, we started from this intuition and

[11] https://github.com/nok/sklearn-porter

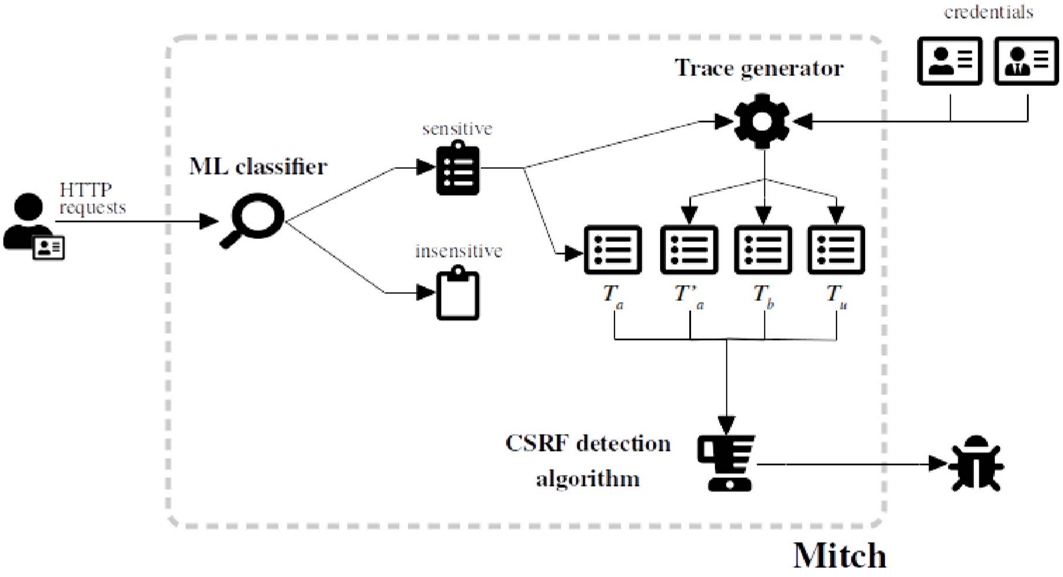


Fig. 6: Mitch architecture

performed preliminary tests on 5 websites to come up with the 1% threshold: we then empirically observed this generalized well to other websites and applications, based on the number of false positives and false negatives. We still acknowledge that different approaches may be worth exploring in the future after a more extensive experimental evaluation.

D. Discussion

Observe that, in principle, the human component of Figure 6 could be entirely replaced by an automated crawler for website navigation. The current Mitch prototype does not include such component for simplicity, but we plan to study its inclusion in future releases to fully automate the CSRF detection process. Moreover, Mitch is designed to be *modular*: its performance can be further improved in different ways, e.g., by improving the ML classifier or by revising the response dissimilarity definition used in the CSRF detection algorithm.

VI. Experimental Evaluation

In this section, we evaluate the effectiveness of Mitch in detecting CSRF vulnerabilities. In particular, we show that the number of false positives and false negatives returned by Mitch is remarkably low and amenable for a practical use.

A. Methodology

Mitch produces a false positive when it returns a candidate CSRF that cannot be actually exploited. This is something relatively easy to detect by manual testing, e.g., using Burp, though this process is tedious and time-consuming. Unfortunately, notice that it is not possible to reliably identify when Mitch produces a false negative, because this would require to know all the existing CSRF vulnerabilities. To estimate this important aspect, we keep track of all the sensitive requests returned by the machine learning classifier embedded into Mitch and we focus our manual testing on those cases. We think this is a very reasonable choice to make the analysis tractable, because we showed in Section IV-D that the classifier performs well using standard validity measures.

Our experimental evaluation does not try to ensure full *coverage* of all the security-sensitive functionalities. Coverage is clearly an important aspect of practical security, yet it is orthogonal to the choice of the machine learning classifier and the CSRF detection algorithm implemented in Mitch, which are the two key components of our approach. Since our current prototype does not include any crawler component, we only performed a relatively short navigation session in all our experiments without any ambition of targeting full coverage, yet doing our best to test a number of significant features. We applied in our experiments the same ethical guidelines reported in Section III-D.

B. Assessment on Existing Websites

To test how effective is Mitch on existing websites, we sampled 20 websites from the Alexa Top 10k ranking. We only considered websites with single sign-on access via a major social network website, so we could leverage just two existing social accounts to perform our security testing. We acknowledge that the use of social single sign-on might have excluded some website categories from our analysis, e.g., e-banking, yet there are clear ethical reasons why such tests have not been performed. Still, in the next section we test Mitch on security-sensitive web applications including e-shops, which we could install and run locally, with positive results.

Overall, Mitch found 191 sensitive requests and reported 47 potential CSRF vulnerabilities: we were able to immediately exploit 35 of them, exposing major security issues in a few

Website	*Sensitive Requests*	*Detected CSRFs*	*fp*	*fn*
9gag.com	10	3	1	0
ask.fm	16	0	0	0
askubuntu.com	16	0	0	0
bombas.com	2	1	0	1
brilio.net	2	1	0	1
eprice.it	11	3	0	3
flixbus.com	4	1	1	0
funnyjunk.com	17	8	2	2
gsmarena.com	3	3	0	0
imdb.com	10	0	0	0
imgur.com	12	3	3	0
indeed.com	8	4	0	0
instructables.com	11	4	0	0
mocospace.com	7	5	2	0
pornhub.com	13	2	1	0
smokecartel.com	5	2	0	0
starnow.com.au	8	4	0	0
tomshardware.com	13	1	1	0
wish.com	11	0	0	0
yelp.com	12	2	1	0
TOTAL	191	47	12	7

TABLE IV: CSRF detection on existing websites

cases. We estimated only 7 false negatives in total, which means that our heuristics are accurate enough to capture most of the vulnerabilities. These numbers lead to the following online validity measures: *precision* $= 0.74$, *recall* $= 0.83$ and $F_1 = 0.78$. The full breakdown on the individual websites is shown in Table IV and commented in the rest of the section.

1) Description of the Attacks: Many of the attacks we found targeted the social functionalities of the websites we tested, like casting votes on public contents, adding or removing items from favorite lists, and posting comments under the identity of the victim. Most of these attacks may thus affect recommender systems, lead to social embarrassment, and compromise user reputation at scale. Even worse, we were also able to find a number of nasty attacks which seriously compromised the website functionality; we responsibly disclosed all the vulnerabilities to the respective website owners. We discuss below the most interesting cases.

a) Bombas: Bombas is an e-commerce website selling socks. It provides a functionality to store a list of shipping addresses to simplify purchases, so that shipping details do not need to be entered for each transaction. The form used to store a new shipping address is vulnerable to CSRF, so an attacker can force any address into the victim's account to hijack deliveries. Notice that the latest added address is the one which is used by default, which makes the attack even worse in terms of practical impact.

Remarkably, Bombas is a customer of Shopify, which is a major e-commerce platform, so this attack may also affect many other websites. We reported the issue to Shopify, which acknowledged the attack and is working on a fix, but marked our report as duplicate due to the existence of a previous independent disclosure.

b) Indeed: Indeed is one of the biggest websites hosting job offers. Registered users can send their CVs and apply to different open positions in the world. We found three CSRF vulnerabilities which give an attacker the ability of fully managing the job offers associated to the account, including the possibility of storing new offers and archiving existing ones. Indeed also suffers from a CSRF vulnerability on the form used to set user preferences, which can be used to severely affect the visibility of job offers. An attacker can exploit this vulnerability to hide job offers, for instance by restricting the search radius and changing the desired publication date for displayed offers.

We find these vulnerabilities particularly interesting, because Indeed is making wide use of anti-CSRF tokens and all the vulnerable forms have their own token. However, it seems that not all the tokens are correctly checked by the website, which may suggest a manual, error-prone placement of the tokens. More generally, this shows that checking the presence of anti-CSRF tokens is not sufficient to say that a website is protected against CSRF, and that the actual website behavior should be tested instead. The security team of Indeed acknowledged the issue and rewarded us $100 for the finding.

c) Starnow: Starnow is an Australian website designed to discover new talents, such as singers and actors. Users who are interested into pursuing an artistic career can register to the website to get access to a number of auditions and job interviews. The first two CSRFs we found allow an attacker to arbitrarily manipulate the watchlists of authenticated users, thus compromising a functionality offered by the website.

There are however two much worse attacks. A CSRF vulnerability affects the form used to store the phone number associated to user profiles: this can be used for scams or to disrupt the functionality of the website, e.g., by making impossible to contact the victim for an audition. It is interesting that the request used to set the phone number contains an anti-CSRF token, which however is not checked by the website: this confirms that this kind of mistakes is not confined to Indeed, but is apparently more widespread.

The last CSRF vulnerability is definitely the most severe one, because it affects the form used to set the email address of user profiles. By exploiting this vulnerability, the attacker can set the victim's email address to her own one and then use the password reset functionality of Starnow to get a fresh password for the victim in her inbox, thus taking possession of the victim's account.

d) Instructables: Instructables is a website where users can upload instructions on how to carry out specific tasks. We found four CSRF vulnerabilities on the website, which grant a wide number of capabilities to the attacker: adding a topic to follow, adding a user to follow, posting comments, and changing the personal email address. The last vulnerability is particularly concerning because the attacker can change the victim's address with her own and claim the victim's password, thus leading to account takeover. The owners of Instructables acknowledged the problem and implemented a fix.

e) Brilio: Brilio is an Indonesian website where people can get paid for the engaging content they post. We confirmed a vulnerability in the user settings page. This page contains, along with the usual personal details, the user payment details in the form of a bank account number. An attacker could thus force the payment details of an unsuspecting victim to her own bank account and redeem the victim's earnings.

2) False Positives: Overall, Mitch reported only 12 false positives in the 20 websites we tested. We think this is a strong result and it was easy to identify these spurious cases by manual testing. Still, it is interesting to investigate the presence of patterns which lead to false positives.

The most troublesome pattern we found is the use of forms to customize user preferences. In these cases, the HTML page returned to Alice and Bob is typically the same even when settings cannot be changed by cross-site requests, because setting pages are often account-independent and failures are only reported occasionally via JavaScript, which is unnoticeable in the static HTML: this scenario led to the introduction of 4 false positives. We then had 2 false positives due to the presence of operations which could be replayed by Alice, but not by Bob: we think these cases are difficult to avoid, because there is no way to understand whether the failure is due to the state of Bob, e.g., the resource accessed by Alice is unavailable to Bob, or to the implementation of anti-CSRF protection. We finally had 3 false positives due to a misclassification of the RF classifier and 3 false positives due to site-specific behaviors which escape our heuristics.

3) False Negatives: We manually investigated all the 144 sensitive requests which were not marked as attacks by Mitch, and we only identified 7 exploitable CSRF vulnerabilities. We only focused on attacks which were quite simple to carry out, i.e., without performing any complicated manipulation of the request parameters, so we do not exclude there might be other vulnerabilities which might be exploited by particularly clever attackers. What we did was just checking manually the outcome of the response received in Bob's session when replaying Alice's requests from a cross-site position, using the HTML elements generated by Mitch to send such requests.

The most problematic pattern leading to false negatives that we found is the use of HTTP requests producing empty responses, irrespective of the outcome of the requested operation. We observed 4 such cases in our experiments, which are bound to escape most forms of black-box detection techniques. As to the other cases, we had 1 failure due to our treatment of JSON responses, 1 failure due to the need of changing the value of a parameter containing an email address to make the attack work, and 1 failure due to the possibility of performing a security-sensitive action in an unauthenticated context, i.e., adding items to the shopping cart.

C. Assessment on Production Software

As a second set of experiments, we decided to run Mitch on the testbed of open-source web applications used to evaluate Deemon [28], which is the only automated detection tool for CSRF vulnerabilities available nowadays. Notice that, since Deemon only works on PHP applications whose source code is available for dynamic analysis, we could not test it on the closed-source websites from our first set of experiments. Out of the 10 applications considered in the original testbed, we were only able to find 3 applications at the same version: Oxid e-shop, Prestashop and Simple Machine Forums. No CSRF vulnerability was detected by Deemon on these applications, according to the experimental evaluation in [28]. The results of the analysis performed by Mitch on the applications in their default configuration are shown in Table V.

Web application	*Sensitive Requests*	*Detected CSRFs*	*fp*	*fn*
Oxid e-shop 4.9.8	21	4	1	0
Prestashop 1.6.1.2	12	1	1	0
SM Forums 2.0.12	9	0	0	0
TOTAL	42	5	2	0

TABLE V: CSRF detection on production software

Mitch was extremely effective on the tested applications, because it reported only 2 false positives and it was able to catch 3 CSRF vulnerabilities on Oxid e-shop which were not reported by Deemon in [28]. These vulnerabilities allow an attacker to corrupt the integrity of the shopping cart, force the use of vouchers and change the preferred payment method. Remarkably, all the corresponding functionalities are supposed to be protected by an anti-CSRF token, which however is apparently not checked by the Oxid back-end. We reported the issues to the Oxid security team, who acknowledged the problem and worked on a fix.

VII. Comparison with Other Tools

In this section, we compare Mitch against other tools which can be used for CSRF detection. We discriminate between academic software based on published research and open-source software freely available on the Web.

A. Academic Software

Deemon [28] is the first research tool for the automated detection of CSRF vulnerabilities and it represents the academic work most closely related to ours. Deemon is based on a security monitor implemented in the PHP interpreter, hence it only works on PHP applications whose source code is available for dynamic analysis, which is a major limitation to its widespread adoption. An important goal of the present work was overcoming this limitation, and it is remarkable that Mitch was able to find vulnerabilities in web applications which were previously analyzed with Deemon without exposing security issues. The main drawback of Mitch with respect to Deemon is its focus on *visibly* sensitive requests, which misses server-side changes with no immediate feedback at the browser side. This limitation is inherent to black-box detection approaches.

CSRF-checker [34] is a black-box detection tool for a class of CSRF vulnerabilities, called *authentication* CSRF, which affect web authentication and identity management. This class of CSRF vulnerabilities enables *login CSRF* attacks [2], which authenticate the victim as the attacker, as well as attacks which allow the attacker to take control of the victim's account. There are a few similarities between CSRF-checker and Mitch, most notably the choice of hijacking HTTP requests sent under one session to another session, but there are major differences as well. CSRF-checker exclusively focuses on a specific class of CSRFs affecting authentication, including login CSRFs, which are not covered by Mitch. Mitch, in turn, targets traditional CSRFs which abuse existing authenticated sessions, hence the overlap between the two classes of detected vulnerabilities is fairly limited. Also, it is important to stress that CSRF-checker is not based on machine learning or automated heuristics, but it simply guides the human tester through the security testing strategy. Human feedback is required both to detect sensitive requests and to detect the success or failure of the performed operations in the two sessions.

B. Freeware and Open-source Software

Penetration testers have been using a range of different tools to detect CSRF vulnerabilities in web applications. Based on an extensive research on blogs, forums and resources for security practitioners, including the OWASP Testing Guide, we classified existing tools in the following categories:

1) *intercepting proxies*: these tools allow penetration testers to intercept and modify arbitrary HTTP traffic, which can be used for an essentially manual detection of web vulnerabilities, including CSRF. Popular tools in this category are Burp, ZAP, and WebScarab;
2) *exploit generators*: these tools simplify the generation of proof of concepts for attack finding, based on human guidance on the set of HTTP requests which need to be tested for CSRF. Examples tools in this category include CSRFTester and pinata-csrf-tool;
3) *web application scanners*: these tools automatically detect a range of web application vulnerabilities, including CSRF, based on different heuristics. Scanners supporting modules for CSRF are Arachni, Skipfish, and w3af.

Our work improves on 1) and 2) by providing effective automated techniques for the detection and the exploitation of sensitive HTTP requests, as opposed to manual investigation and testing. The most important advances over 3) are instead the use of machine learning for sensitive request detection, a more sophisticated CSRF detection algorithm and a systematic evaluation of the performance of our detection tool, based on the analysis of false positives and false negatives produced on real web applications. Remarkably, we noticed important design limitations in the opensource tools we analyzed, which significantly downgrade their accuracy.

For example, Arachni only detects CSRF vulnerabilities on forms requiring an authenticated context, hence it does not capture CSRF attempts via links or AJAX. The rationale behind this choice is likely the complexity of detecting sensitive HTTP requests, which forced the developers of Arachni to limit their tool to HTML elements which are potentially dangerous, yet easy to catch syntactically (forms). It is instructive that w3af suffers from a somewhat opposite design choice: since any request which includes cookies and parameters is deemed as potentially sensitive by w3af, the tool is affected by a very large number of false positives, which led to the opening of an issue on GitHub where a major redesign of the tool is advocated[12]. Other issues we found are related to the choice of deeming secure any HTTP request which includes an anti-CSRF token, while we observed several cases where tokens are not checked at the web application back-end, and to the use of just a single authenticated session, which loses precision when user-dependent secrets happen to thwart CSRF attempts. In the end, preliminary tests with existing web application scanners on simple examples returned a high number of false positives and false negatives, which is in line with the findings of previous research work which showed the ineffectiveness of such scanners for CSRF detection [3], [15].

VIII. Related Work

In the previous section, we discussed existing approaches to CSRF detection, i.e., tools designed to detect CSRF vulnerabilities in web applications. Here we report on additional related work on CSRF defenses and machine learning for security. As to other serious threats to web session security, we refer to a recent survey on the topic [8].

A. Cross-Site Request Forgery

Robust defenses against CSRF were first proposed by Barth, Jackson and Mitchell in their seminal paper [2]. Their proposals still represent the state of the art to protect web applications against CSRF, which typically resort to tokenization and referrer checking to prevent such attacks. However, the research community proposed also a number of alternative solutions against CSRF, which did not find wide practical adoption. In particular, it is worth mentioning several browser-side defenses like RequestRodeo [21], BEAP [26], and CsFire [30], [31], which share the idea of stripping session cookies from cross-site requests to prevent CSRF. Other similar solutions instead involve the browser's user in the loop when suspicious cross-site requests are detected [32]. These browser-side defenses are useful to protect the victims of a CSRF attempt, but they cannot assist security-conscious web developers who want to identify potential room for CSRF in their web applications.

It is worth noticing that existing browser-side defenses are all based on heuristics to detect sensitive requests, though different heuristics are used by different tools. In Section IV-E we showed that previously proposed syntactic detection heuristics [26], [30] behave poorly with respect to the classifiers trained in the present work; similarly, heuristics based on runtime checks [31] proved insufficient to support legitimate functionality of normal browsing according to previous research [13]. Indeed, the misclassification rate of the heuristics represented a major obstacle to the usability of these browser-side solutions, which encouraged the development of server-guided protection techniques like Allowed Referrer Lists [13] and micro-policies [7]. We believe that our supervised learning approach to the detection of sensitive HTTP requests could be integrated in existing browser-side defenses against CSRF to reduce their misclassification rate and improve their usability. However, we remark that adversarial machine learning techniques would be needed to train a classifier which is resilient to evasion attempts [1]. This was not required for Mitch, which is a CSRF detection tool designed for security testers and not a

[12] https://github.com/andresriancho/w3af/issues/120

defense mechanism against CSRF like those mentioned above. This implies that the audience of the two tools is different: for example, false positives can be accepted by security testers, but they cannot be accepted by browser users, because they would break legitimate website functionality. Also, security testers may be willing to let Mitch replay the same HTTP request multiple times to collect additional data for CSRF detection, but this approach would not be appropriate for browser-side protection, given its overhead on web traffic and performance.

Finally, it is worth noticing the existence of model-based techniques to verify protection against CSRF attacks in web applications, e.g., via model-checking [29]. These approaches operate on abstract models of the analyzed web applications, which require access to their source code or extensive manual testing to come up with a faithful representation. They thus work better on relatively small systems with a clear formal specification, like web protocols.

B. Machine Learning for Security

Machine learning has found a wide number of applications to computer security, for instance in intrusion detection systems [33], malware detectors [24], and spam filters [19]. In the context of web security, supervised learning techniques have been proposed to automatically detect cookies which are used for authentication purposes [9], [10]. This is useful to instruct the browser into applying stronger security policies on them, e.g., to prevent their improper disclosure.

IX. Conclusions and Future Work

Mitch is the first machine learning solution for the black-box detection of CSRF vulnerabilities. It exploits supervised learning techniques to accurately identify HTTP requests which require protection against CSRF (sensitive requests) and relies on heuristics to find attacks abusing them. We experimentally showed that Mitch produces a small number of false positives in practice, and that it is effective enough to expose new CSRF vulnerabilities in existing websites and production software, some of which escaped previous approaches. By using Mitch, we observed that web application developers are aware of the dangers of CSRF and typically try to implement appropriate protection mechanisms, yet they accidentally leave room for attacks, e.g., because they forget to check anti-CSRF tokens which are sent back to the web application.

There are a number of avenues for future work. Adversarial learning is an active research area whose main goal is designing classification algorithms which are robust to the presence of attackers who actively try to fool them into misprediction [1]. Our set of features and classifiers are not intended to be resilient to adversarial manipulations. Indeed, this is out of scope for our present endeavors, because our classifiers are intended to assist penetration testers in finding CSRF vulnerabilities and are *not* designed for CSRF attack detection. We therefore consider adversarial detection of CSRF attacks as an interesting direction for future work, i.e., developing online detection systems for CSRF attacks which are robust against sophisticated attempts to mask sensitive requests as insensitive (or vice versa). In addition, we plan to further improve our classifiers and heuristics to reduce even more the number of false positives and false negatives reported by Mitch. In this respect, we count on extracting new features from HTTP requests that turn into useful predictors (e.g., more advanced textual signals). Finally, we would like to leverage Mitch to carry out a large-scale detection of CSRF vulnerabilities in the wild, so as to get a better understanding of the severity of CSRF attacks on the current Web. This would require the construction of an automated crawler component to fully mechanize the security analysis.

References

[1] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, “The security of machine learning,” *Machine Learning*, vol. 81, no. 2, pp. 121–148, 2010.

[2] A. Barth, C. Jackson, and J. C. Mitchell, “Robust defenses for cross-site request forgery,” in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 75–88.

[3] J. Bau, E. Bursztein, D. Gupta, and J. C. Mitchell, “State of the art: Automated black-box web application vulnerability testing,” in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, 2010, pp. 332–345.

[4] A. Bhandare, M. Bhide, P. Gokhale, and C. Rohan, “Applications of convolutional neural networks,” *International Journal of Computer Science and Information Technologies*, vol. 7, pp. 2206–2215, September–October 2016.

[5] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[6] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[7] S. Calzavara, R. Focardi, N. Grimm, and M. Maffei, “Micro-policies for web session security,” in *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, 2016, pp. 179–193.

[8] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, “Surviving the web: A journey into web session security,” *ACM Comput. Surv.*, vol. 50, no. 1, pp. 13:1–13:34, 2017.

[9] S. Calzavara, G. Tolomei, M. Bugliesi, and S. Orlando, “Quite a mess in my cookie jar!: leveraging machine learning to protect web authentication,” in *23rd International World Wide Web Conference, WWW ’14, Seoul, Republic of Korea, April 7-11, 2014*, 2014, pp. 189–200.

[10] S. Calzavara, G. Tolomei, A. Casini, M. Bugliesi, and S. Orlando, “A supervised learning approach to protect client authentication on the web,” *TWEB*, vol. 9, no. 3, pp. 15:1–15:30, 2015.

[11] G. C. Cawley and N. L. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” *Journal of Machine Learning Research*, vol. 11, pp. 2079–2107, Aug. 2010.

[12] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[13] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang, “Lightweight server support for browser-based CSRF protection,” in *22nd International World Wide Web Conference, WWW ’13, Rio de Janeiro, Brazil, May 13-17, 2013*, 2013, pp. 273–284.

[14] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, Oct. 2012.

[15] A. Doupé, M. Cova, and G. Vigna, “Why johnny can’t pentest: An analysis of black-box web vulnerability scanners,” in *Detection of Intrusions and Malware, and Vulnerability Assessment, 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings*, 2010, pp. 111–131.

[16] T. Fawcett, “An introduction to roc analysis,” *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[17] J. H. Friedman, “Stochastic gradient boosting,” *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, Feb. 2002.

[18] I. Guyon and A. Elisseeff, “An introduction to variable and feature selection,” *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[19] T. S. Guzella and W. M. Caminhas, "A review of machine learning approaches to spam filtering," *Expert Syst. Appl.*, vol. 36, no. 7, pp. 10 206–10 222, 2009.

[20] T. K. Ho, "Random decision forests," in *Proceedings of the Third International Conference on Document Analysis and Recognition - Volume 1*, ser. ICDAR '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 278–282.

[21] M. Johns and J. Winter, "Requestrodeo: client side protection against session riding," in *Proceedings of the OWASP Europe Conference, refereed papers track, Report CW448*, 2006, pp. 5–17.

[22] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," in *The Handbook of Brain Theory and Neural Networks*, M. A. Arbib, Ed. Cambridge, MA, USA: MIT Press, 1998, pp. 255–258.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[24] C. LeDoux and A. Lakhotia, "Malware and machine learning," in *Intelligent Methods for Cyber Warfare*, 2015, pp. 1–42.

[25] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, "Feature selection: A data perspective," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 94:1–94:45, 2017.

[26] Z. Mao, N. Li, and I. Molloy, "Defeating cross-site request forgery attacks with browser-enforced authenticity protection," in *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, 2009, pp. 238–255.

[27] V. L. Parsons, *Stratified Sampling*. American Cancer Society, 2017, pp. 1–11. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat05999.pub2

[28] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with dynamic analysis and property graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1757–1771.

[29] M. Rocchetto, M. Ochoa, and M. T. Dashti, "Model-based detection of CSRF," in *ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings*, 2014, pp. 30–43.

[37] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *IEEE Symposium*

[30] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen, "Csfire: Transparent client-side mitigation of malicious cross-domain requests," in *Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, 2010, pp. 18–34.

[31] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens, "Automatic and precise client-side protection against CSRF attacks," in *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, 2011, pp. 100–116.

[32] H. Shahriar and M. Zulkernine, "Client-side detection of cross-site request forgery attacks," in *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, 2010, pp. 358–367.

[33] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, 2010, pp. 305–316.

[34] A. Sudhodanan, R. Carbone, L. Compagna, N. Dolgin, A. Armando, and U. Morelli, "Large-scale analysis & detection of authentication cross-site request forgeries," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017, pp. 350–365.

[35] G. Tolomei, F. Silvestri, A. Haines, and M. Lalmas, "Interpretable predictions of tree-based ensembles via actionable feature tweaking," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, 2017, pp. 465–474.

[36] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, "CAPTCHA: using hard AI problems for security," in *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, 2003, pp. 294–311.
on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA, 2012, pp. 365–379.

[38] H.-F. Yu, F.-L. Huang, and C.-J. Lin, "Dual coordinate descent methods for logistic regression and maximum entropy models," *Machine Learning*, vol. 85, no. 1, pp. 41–75, Oct 2011.