Università Ca' Foscari di Venezia
Dipartimento di Scienze Ambientali, Informatica e Statistica

Corso di Laurea Magistrale in Informatica - Computer Science

Tesi di laurea magistrale

Understanding Machine Learning Effectiveness to Protect Web Authentication

Candidato:
Andrea Casini
Matricola 819522

Relatore:
Ch. Prof. Salvatore Orlando

Correlatori:
Dott. Stefano Calzavara
Dott. Gabriele Tolomei

Anno Accademico 2013–2014

I would like to dedicate this work to my ever-supporting family.

ABSTRACT

Cookie-based web authentication is the most widespread practice to maintain the user's web session. This mechanism is, inherently, subject to serious security threats: an attacker who acquires a copy of cookies containing authentication information may be able to impersonate the user and conduct a session on their behalf. Recently, browser-side defenses have proven to be an effective protection measure against these types of attacks. In existing approaches, all such defenses ultimately rely on empirical client-side heuristics to automatically detect authentication cookies to eventually protect them against theft or otherwise unintended use.

In this thesis, we build upon a conference paper published at WWW' 14 [Calzavara *et al.*, 2014] to overcome its limitations. Specifically: (1) the results of such a document are based on a gold set of only 327 cookies collected from 70 websites. In this work, we extend our analysis to a much larger dataset of approximately 2500 cookies gathered from 220 popular website according to the Alexa ranking. (2) we implement a faster and more accurate authentication token detection method for which our gold set is constructed, including full Javascript support. (3) we confirm a popular literature assumption according to which the number of authentication cookies registered by Javascript is negligible. (4) we formalize a novel measure of protection used to evaluate further effectiveness of previous heuristics from the literature, as well as our approach. (5) we adopt a different machine learning approach to deal with new challenges that, mainly, arise from a larger dimension of the dataset and from the distribution of its instances.

The results of our work, ultimately, provide a more in-depth sight of how web authentication is implemented in practice and what kind of security measures are adopted throughout the Web.

ACKNOWLEDGMENTS

I wish first of all to thank Prof. Salvatore Orlando, Stefano Calzavara and Gabriele Tolomei for their invaluable aid during the writing of this work, the detailed explanations, the patience and the precision in the suggestions, the supplied solutions, the competence and the kindness. Thanks also to my classmates Gianpietro Basei and Mauro Tempesta, who have discussed with me, giving precious observations and good advices.

Venice, october 2014 Andrea Casini

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

INTRODUCTION

The HTTP(S) protocol that governs the World Wide Web is, by design, *stateless*. Nowadays, websites have become complex applications and, mostly, require to keep track of the user across multiple HTTP(S) requests. The most widespread workaround to this flaw rely on cookies: key-value pairs which are sent by the server to the user's browser. Web authentication is the most common example of cookies use. Once a user authenticates, the server sends one or multiple cookies to her browser, creating a user-specific *session*. We will refer to such cookies as *authentication cookies*, to emphasize the fact that they contain the information necessary to keep the user authenticated into the website.

This cookies are, inherently, the primary target for a widespread class of attacks aimed at hijacking the user's session: attackers, who acquires a copy of authentication cookies, can impersonate the user and exploit her privileges in the authenticated session.

In this work, we explore the World Wide Web, investigating how web authentication is deployed and how effective is a client-side protection mechanism at preventing session hijacking. Specifically, our document is organized as follows:

THE FIRST CHAPTER provides the reader with the basic concepts on which this work is funded. We describe how HTTP(S) protocol handle the requests and the responses between a client and a server. We introduce what a web cookie is and how it is employed in the web authentication. Finally, we investigate the weakness of such a mechanism, exploring the typical attacks that hackers adopt to hijack the user's session.

THE SECOND CHAPTER is dedicated to the construction of a gold set of cookies that will be used to assess some literature client-side defenses and to devise a novel authentication cookies detector. The challenge here is crawling the Web, gathering cookies from different websites and, finally, detecting the authentication tokens; a fundamental concept that we will be covered in detail.

THE THIRD CHAPTER contains the descriptions and the assessments we perform over some literature client-side defenses. The key idea underlying such protection measures is to apply the security practices neglected by the server by detecting authentication cookies at the client-side, and enforcing a more conservative browser behavior when accessing them.

This process ultimately rely on an authentication cookie detector, i.e., a heuristics which tries to automatically identify the cookies associated with the user credentials among all the cookies stored in the browser, with no support by the user or the remote server. In this chapter, we evaluate the performance of such a methodology according to how good is the balance between *security* and *usability*. On the one hand (security) such detector should not miss any authentication cookie, as any miss may leave room for attacks. At the same time (usability), they should not over-approximate too much the real set of authentication cookies, since the conservative security policy applied for them may harm the user experience.

THE FOURTH CHAPTER introduces a novel machine learning approach to detect authentication cookies. Devising such a detector is not trivial at all. We have to deal with the typical challenges posed by a supervised learning task specifically: *feature extraction*, *feature selection*, *overfitting* and, ultimately, the *class imbalance problem*. We conduct several experiments comparing our approach with the existing solutions, we evaluate in the third chapter. Interestingly, it turns out that our method outperforms all the other providing a good tradeoff between security and usability.

CONTRIBUTIONS

This work is build upon a conference paper published at WWW' 14 [Calzavara *et al.*, 2014]. The additional contributions we provide, are the following:

1. All the experiments are performed on a much larger gold set of cookies (2,546 vs. 327) collected from many more websites (220 vs. 70). This allows us to give further and more significant evidence of the problems we identified in our previous work and of the effectiveness of the solutions we proposed therein;

2. Building a larger gold set poses new challenges both in terms of computational efficiency and in terms of implementation choices. We discuss a non-trivial optimization of the original algorithm, which runs in linear time (with respect to the number of cookies) for the large majority of the websites and is much faster in practice than the original solution. The new implementation is based on the `Selenium` Python package rather than `Mechanize`, since the lack of support for JavaScript in `Mechanize` makes many websites not working. By using `Selenium`, we are able to include in the gold set all the cookies which are set by JavaScript, which we excluded from our previous study;

3. We experimentally confirm a popular assumption from the literature: the number of authentication cookies which is set by JavaScript is negligible, since we observe that only 4 out of 220 websites (1.8%) use JavaScript to set an authentication cookie;

4. We formalize a novel measure of protection, which we use to evaluate further the effectiveness of previous heuristics from the literature, as well as our approach;

5. The larger gold set we consider in this work is skewed: approximately, only 1 out of 7 cookies is used for authentication purposes. Being the gold set skewed, it is more difficult to train an effective classifier. We then revise and make more systematic the machine learning approach to overcome this new challenge and solve the performance problems we observed.

1 BACKGROUND

This chapter is intended to provide the reader with the basic concepts that will be discussed throughout this thesis. In the first section, a brief introduction of the HTTP protocol is given: we explain how do a client and a server communicate and what are the rules and responsibilities between those two components.

The second section introduces *Web cookies*, pieces of information that the server and client pass back and forth. The amount of information is usually small, and its content is at the discretion of the server. Cookies fulfill a fundamental task for the usability of modern websites: they provide a memory mechanism between Web pages, keeping track of sensible information about the user e.g. the credentials to authenticate into a certain website. For this reason cookies are often a controversial topic when discussing about privacy and security of the users [Kristol, 2001]. Stealing cookies is, in fact, a serious *security threat*: an attacker who acquires a copy of an authentication cookie may be able to use it to impersonate the user and conduct a session on their behalf. The most common attacks aimed at cookies theft are discussed in the final section together with some solutions to prevent them.

1.1 CLIENTS AND SERVERS

The Hypertext Transfer Protocol (HTTP) provides the foundation for the Web: it handles the requests and the responses between a *client* and a *server*. Those two components must follow very different rules and procedures dictated by such protocol. In a simple Web session, the Web browsing agent is an HTTP client, while the system hosting the website acts as an HTTP server [Fielding *et al.*, 1999].

Clients and servers have specific roles.

The most obvious difference between HTTP clients and servers is responsibility for initiating communication. Only clients can do that. A server may provide lot of information and perform many functions, but it does something only when asked to do so by a client [Thomas, 2001]. In other words a client make a request and the communicating server reacts with a response.

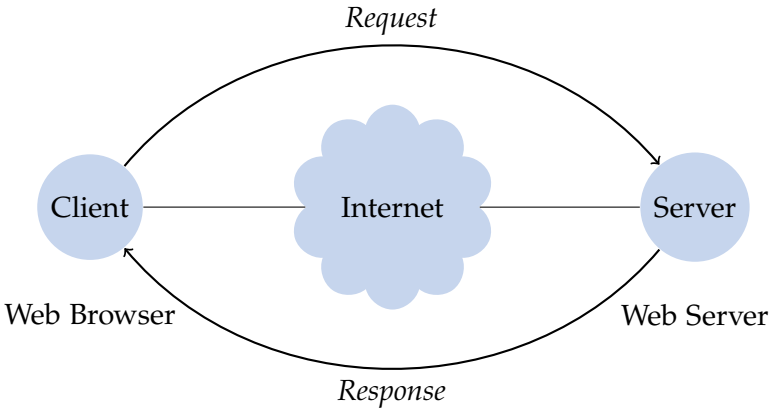

Figure 1: The client begins a communications exchange by sending a request to a server. The server simply responds to client requests. It does not initiate communications on its own.

Basic HTTP operations.

The HTTP protocol defines some basic operations each of them as a result of a user action. The simplest of all is `GET`. It is how a client retrieves an object from the server. On the Web, browsers request a page from a Web server with a `GET`. If the server can return the requested object, it does so in its response along with a status code indicating the success (otherwise failure) of the interaction.

While `GET` allows a server send information to a client, the `POST` operation provides a way for clients to send information to servers. This type of operation is commonly adopted by web browser to send forms to web servers. For example when a user submit a login form or by clicking a search button the browser sends a POST request to the server including the information the user has provided in the form.

1.2 COOKIES

The nature of the Web is stateless.

The HTTP protocol normally operates as if each client request is *independent* of all others. The server responds to any request strictly on the merits of that request, without reference to other requests from the client or any other one. This type of operation is known as **stateless** because the server does not have to maintain the state of its clients.

Keeping track of the client state requires server resources (memory, computational power, etc.), therefore stateless operation is usually preferable. In some applications, however, the server needs to keep some state information about each of its clients. For instance users that successfully authenticate into a website should not have to log in again every time to view different pages on the site.

Cookies are like stickers stuck onto users by servers.

The most widely deployed mechanism for maintaining client state are HTTP **cookies**. A cookie is basically a small piece of data sent from a server and stored in a user's web browser while the user is browsing the website hosted by such server. A server creates cookies when it wants to track the state of a client, and it returns those cookies to the client in its response. Once the client receives a cookie, it can include the cookie in subsequent requests to the same server. Cookies were first developed by Netscape but now are supported by all major browsers [Thomas, 2001].

Cookies consist of a set of attributes listed in the following table. It is up to the server to choose the values for such attributes.

Table 1: Cookies attributes

Attribute	Status	Notes
Name	Required	An arbitrary name of the cookie assigned by the server.
Value	Required	An arbitrary value of the cookie assigned by the server.
Domain	Optional	The domain that the cookie is available to. Setting the domain to `www.example.com` will make the cookie available in the www subdomain and higher subdomains. Cookies available to a lower domain, such as `example.com` will be available to higher subdomains, such as `www.example.com`.
Path	Optional	The URLs on the server to which the cookie applies. If set to "/", the cookie will be available within the entire domain. If set to "/foo/", the cookie will only be available within the "/foo/" directory and all sub-directories such as "/foo/bar/" of domain. The default value is the current directory that the cookie is being set in.
Secure	Optional	Instructs the client to only transmit the cookie with a secure HTTPS connection;

Table 1: continues onto the next page

Table 1: continues from the previous page

Attribute	Status	Notes
Expiry	Optional	The lifetime of the cookie. This is a Unix timestamp so is in number of seconds since the epoch. If set to 0, or omitted, the cookie will expire at the end of the session (when the browser closes).
HttpOnly	Optional	When true the cookie will be made accessible only through the HTTP protocol.. Therefore such cookie is not accessible through non-HTTP methods such as calls via Javascript (e.g. `document.cookie`).

Table 1: end of table

When the server has omitted one of the optional attributes, the client supplies default values. For example if the domain of a cookie is missing then the browser will set the domain name of the server that supplied such cookie.

Types of cookies: session and persistent.

Cookies can be broadly classified into two types: *session cookies* and *persistent cookies*. A session cookie is a temporary cookie that keeps track of settings and preferences as a user navigates a site. A session cookie is deleted when the user exits the browser. Persistent cookies can live longer; they are stored on disk and survive browser exits and computer restarts. Persistent cookies often are used to retain a configuration profile or login name for a site that a user visits periodically.

1.3 WEB AUTHENTICATION

If the World Wide Web were nothing more than a collection of static information, then securing the Web's protocols would be less important. However, with the growth of electronic commerce and the extension of HTTP to critical environments outside of the Web, adding security to HTTP is critical for many applications. Web authentication allows the communicating parties to verify each other's identity, to ensure the privacy of their communication, and to protect their messages from modification or corruption.

The authentication of a user into a website is the most common example about the cookies usage. The mechanism of a cookie-based authentication is described in the following sequence diagram.

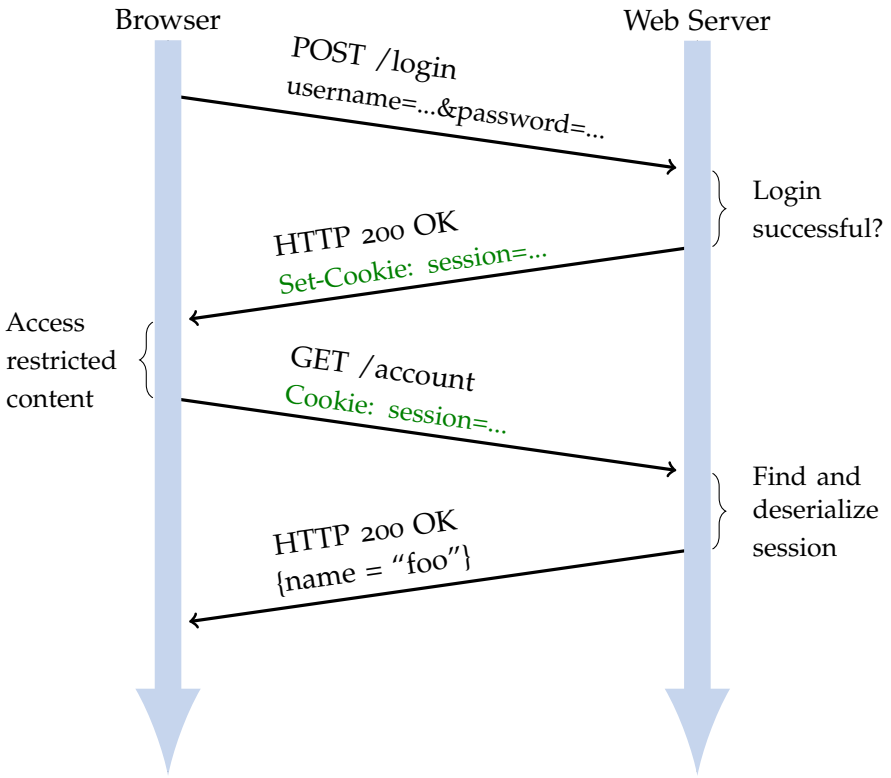

Figure 2: Sequence diagram: cookie-based authentication

From Figure 2, we recognize four phases:

1. *request*: The request made by the client for some restricted document;
2. *challenge*: The server rejects the request, indicating that the user needs to provide her username and password;
3. *authorization*: The client retries the request, attaching authorization information specifying username and password;
4. *success*: If the authorization credentials are correct, the server returns the document with status 200.

The basic authentication scheme is considered to be not secure, as the username and password travel the network in an unencrypted form. For this reason more complex authentication schemes, like Digest authentication or Kerberos, are employed in order to meet the need of security required by the World Wide Web [Franks *et al.*, 1999].

Although many well-studied techniques exist for authentication, websites continue to use extremely weak authentication schemes, especially in non-enterprise environments such as store fronts [Fu *et al.*, 2001]. These weaknesses often result from careless use of authenticators within Web cookies.

1.4 SESSION SECURITY

Cookie-based session are exposed to serious *security threats* as revealing an authentication cookie provides an attacker with full capabilities of impersonating the user identified by such cookie. The problem is so serious that browsers provide cookies with the `Http-Only` and `Secure` flags to protect them from unintended access by scripts injected within HTML. In this section, we are going to explore the most common attacks aimed at hijacking the user web session.

1.4.1 Session Sniffing

The unencrypted traffic exchanged between the browser and the server may be fully inspected by a network attacker. This fact makes room for a variety of network attacks that intercept sensitive information. Figure 3 shows a typical *eavesdropping* attack: the attacker uses a network sniffer to capture a valid session token (e.g. a cookie) called "Session ID", after that he uses the valid token session to gain unauthorized access to the Web Server [Owasp, 2013b].

Eavesdropping is a way of intercepting the information exchange between the client and the server but there exists other network attacks worthy to mention specifically the *man-in-the-middle* and the *man-in-the-browser* attacks [Callegati *et al.*, 2009].

Adopting HTTPS connections to encrypt network traffic would provide an effective countermeasure against these kind of attacks, however protecting authentication cookies against improper disclosure is tricky. In fact, modern Web browsers attach by default the cookies registered by a given domain to *all* the HTTP(S) requests to that domain. Therefore, unless some protection mechanism is adopted, loading a page over HTTPS may still leak authentication cookies, whenever the page retrieves additional contents (e.g. images) over an HTTP connection to the same domain.

The authors of [Bugliesi *et al.*, 2014] demonstrates that these type of attacks can be effectively mitigated at client-side by applying the `Secure` flag to the authentication cookies.

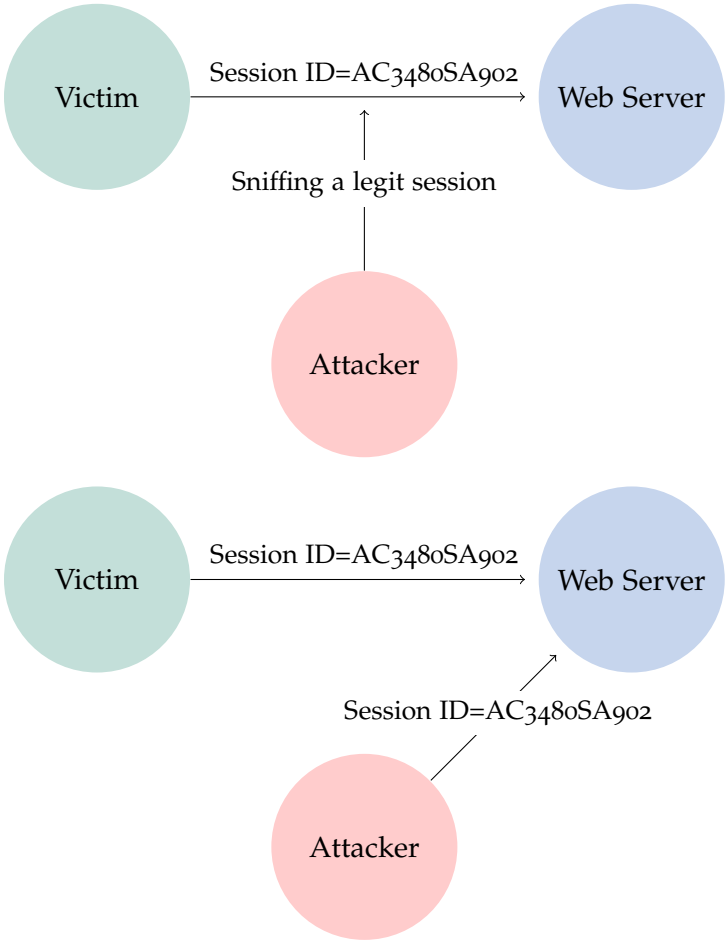

Figure 3: Sniffing the session token to hijack the session

1.4.2 XSS Attacks

Web browsers implement a simple protection mechanism based on the so-called "same-origin policy" whereby cookies registered by a given domain are made accessible only to scripts retrieved from that same domain. Unfortunately, the same-origin policy may be circumvented by a widespread form of code injection attacks known as *cross-site scripting* (XSS). Cross site scripting is an attack technique that forces a website to display malicious code, which then executes in a user's web browser [Grossman, 2007].

XSS circumvent the same-origin policy.

For example, an attacker can send a crafted link to the victim with malicious JavaScript, when the victim clicks on that link, the JavaScript code will run and complete the instructions written by the attacker. In order to steal the victim's session such script can retrieve the Javascript object `document.cookie` containing the authentication cookies and leak them to the attacker's website.

The script reported in Listing 1 creates an image DOM object (row 2) and since the JavaScript code executed within the victim context, it has access to the cookie data i.e. `document.cookie`. The image object is then assigned an off-domain URL to `http://attacker/` appended with the Web browser cookie string where the data is sent (row 3).

Listing 1: Example of an XSS attack script

```
1 <script>
2 var img = new Image();
3 img.src = "http://attacker/" + document.cookie;
4 </script>
```

The correct use of `Http-Only` flag can prevent XSS attacks.

XSS attacks are so widespread that Microsoft in 2002 attempt to mitigate this issue by introducing the `Http-Only` flag to indicate cookies which should not be made accessible on the client [Owasp, 2013a]. If a browser that supports `Http-Only` detects a cookie containing the `Http-Only` flag, and client side script code attempts to read the cookie, the browser returns an empty string as the result. That causes the attack to fail by preventing the malicious code from sending the data to an attacker's website.

1.4.3 Session Fixation

Many efforts have been made in the field of web session security to prevent the attacker from obtaining, either intercepting, predicting or brute-forcing, a session identifier issued by the Web server [Kolšek, 2002]. However, one possibility has been so far ignored: the attacker may fix a session ID to the user's browser, thereby forcing the browser into using a chosen session. This type of attacks are called *session fixation attacks*, because the user's session ID has been fixed in advance instead of having been generated randomly at login time. A simple example of session fixation attack is given in Figure 4.

In this scenario the attacker establishes a legitimate connection with the web server (1) which responds by issuing a session ID (2) then, the attacker sends a link with the established session ID to the victim (3) trying to lure her into clicking on it. The user clicks on the link which opens the server's login page in her browser (4). The Web server application sees that a session was already established and thus a new one need not to be created. Finally, the user provides her credentials to the login form and the server grant access to her account (5). However, at this point, knowing the session ID, the attacker can also access the user's account hijacking the session (6).

A simple (and recommended) server-side practice to prevent session fixation attack consists in generating a fresh session ID whenever the user authenticates into the website. In this way an attacker is not able to hijack the session since the session ID differs from the fixated one. It is also possible to significantly reduce this type of attack at the client-side by requiring that authentication cookies attached to HTTP(S) requests are only registered through HTTP(S) headers [De Ryck *et al.*, 2012].

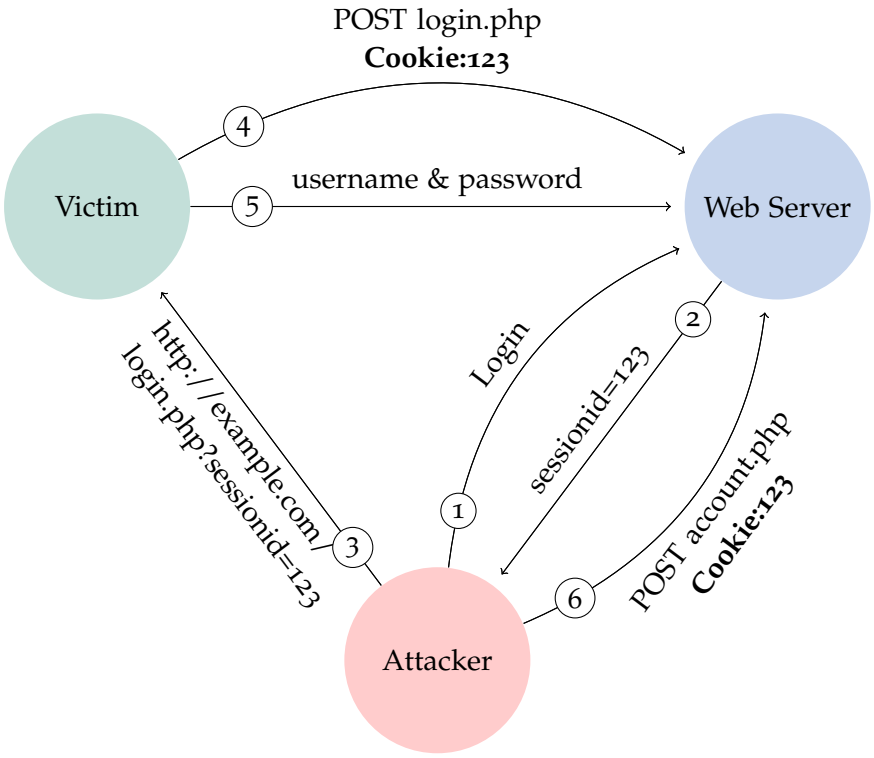

Figure 4: Simple example of a session fixation attack

2 A GOLD SET OF COOKIES

In this chapter, we explain how the construction of the gold set of cookies has been conducted. Our goal is to collect as many cookies as possible identifying the authentication ones: cookies registered by the server that contains the necessary information to keep the user's authenticated session.

We discover that modern websites adopt different authentication schemes. Specifically, one of our most interesting finding is that servers usually registers multiple authentication cookies in the user's browser, each with different unconventional names and values. Therefore, we claim that each website presents a specific **authentication token**, a notion that we will define in detail later on.

Building a gold set of cookies is a typical Web mining task and it consists of two phases:

i. collect cookies from different websites;

ii. assign a binary label to distinguish the authentication cookies from the other ones.

The first step require us to sign up in different websites to create a personal account. We choose for our analysis the top 500 popular websites from the `Alexa` ranking [Alexa, 2014]. Unfortunately, largely adopted security measures, like *captchas*[1], disallow us from automating this procedure. Therefore, we do have to register to each website manually.

As far as the second step is concerned, we implement a tool that, given the user credentials for a specific website, is able to automatically login, gather the cookies and detect the authentication tokens. The implementation of such framework is full of challenges that we will discuss in Section 2.3.

1 an automated test that humans can pass but computer programs can't.

2.1 AUTHENTICATION TOKENS

In Section 1.3, we study how cookie-based authentication works. However, websites adopt different authentication schemes. In particular, we discover that web servers may register group, or even different groups, of authentication cookies, to maintain the authenticated session of a specific user.

The authors of [Calzavara *et al.*, 2014] identify these groups of cookies as authentication tokens. The paper gives a strict definition of this concept:

Definition 1 (Authentication Token)**.** *Let S be a server and $\mathcal{C}$ the set of cookies it sends to the browser B upon login. We say that $A \subseteq \mathcal{C}$ is an* authentication token *for S if and only if the following conditions hold:*

i. authentication*: S authenticates B for any request including all the cookies in A;*

ii. minimality*: S does not authenticate B for any request including only cookies in $A' \subset A$.*

A cookie c is an authentication cookie iff there exists an authentication token A such that $c \in A$.

This notion is better explained with an example. Consider one of the most popular email service: `gmail.com`, known also as Google Mail. Once you create your `gmail` account you can take a look on the cookies that the server has sent to you. At the time when this document is written, `gmail.com` present the following cookies named: `NID`, `SID`, `HSID`, `SSID`, `APISID`, `SAPISID`, `OGPC`, `PREF`. Each of them with a specific set of attributes. If you enumerate all possible combinations of cookies and separately present each of them to the server you will notice that only the combinations containing the cookies `HSID`, `SSID`, `NID` will keep you inside the authenticated session, while all the others will kick you out. Consequently, we conclude that `HSID`, `SSID`, `NID` is the unique authentication token for `gmail.com` and if you delete one of these cookies you will be logged out.

Websites may have multiple, possibly overlapping authentication tokens.

Notice that, according to Definition 1, websites may admits multiple (even overlapping) authentication tokens. For example `amazon`, the popular electronic commerce site, presents two different authentication tokens of size 1 and 2 respectively: `x-main` and `session-id`, `ubid-main` are the cookies' names composing these two tokens. Any of the two is enough to keep the user's session with the website.

This authentication mechanism must be well designed and secured against cookies theft since if an attacker wants to take the user's identity, then it is enough to steal just one authentication token. Therefore, a website is considered not vulnerable if its authentication tokens are properly secured. A definition follows:

Definition 2 (Vulnerability)**.** *An authentication token A is* vulnerable *if and only if every cookie $c \in A$ is known to the attacker.*

Therefore, a client-side defense for cookie-based session is effective whenever its authentication cookie detector is able to find at least one authentication cookie for each authentication token. If this minimal set of authentication cookies is protected it is guaranteed that the website is resistent against any session hijacking attack.

One basic way to protect a cookie against session hijacking attack is by setting the `Http-Only` flag. The idea of `Http-Only` cookies is to denote cookies that are not visible through the DOM. `Http-Only` cookies were introduced by Microsoft in 2002 and first implemented in Internet Explorer 6. To mark a cookie as `Http-Only`, the `Http-Only` attribute is added to the cookie field. A browser that supports `Http-Only` cookies sends the cookies back in HTTP request headers, but will not permit a script to access an `Http-Only` cookie. As a result, even if a cross-site scripting (XSS) flaw exists, and a user accidentally accesses a link that exploits this flaw, the browser will not expose the cookie to a third party.

`Http-Only` cookies mitigate session hijacking attacks.

However, it is important to find a good **tradeoff**: on the one hand *(security)* `Http-Only` cookies prevent XSS attacks on the other hand *(usability)* they may break the functionality of the website [Zhou and Evans, 2010]. For instance, consider a cookie that keeps in memory a shopping cart and suppose that the hosting website access it through Javascript. If this cookie is erroneously identified as an authentication cookie and marked as `Http-Only` then the website will not be able to keep track of the user purchases.

Security and usability must be appropriately balanced.

2.2 BUILDING THE GOLD SET

Let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be the set of n cookies that server S sends to browser B upon authentication. To build our dataset we have to identify a *labelling function* $\hat{f} : \mathcal{C} \to \{0, 1\}$ such that:

$$\hat{f}(c_i) = \begin{cases} 1, & \text{if } c_i \text{ is an authentication cookie} \\ 0, & \text{otherwise} \end{cases}$$

Discovering such a function is a heavy computational task. We have to enumerate all the possible combinations sets of $\mathcal{C}$ and find the one(s) that authenticates B into server S. Websites typically sends 10 cookie upon the browser, requiring us to check over 2^{10} combinations. It may look feasible but note that each iteration is a request sent to the server and the time spent to process it depends on the traffic and the speed of the network. Furthermore, there may exists some security mechanism that block the client access if the number of requests in a specific period of time exceed a server-defined threshold. Therefore, we need to develop a method able to detect authentication tokens while keeping the number of requests **as low as possible**.

Consider the following set of cookies $C = \{a, b, c\}$ stored in the browser B. To fulfill our goal we have to explore $\mathcal{P}(C)$, the powerset of C. At the end of the process, we will found one or multiple authentication tokens. Figure 5 shows a graph representation of the search space $\mathcal{P}(C)$.

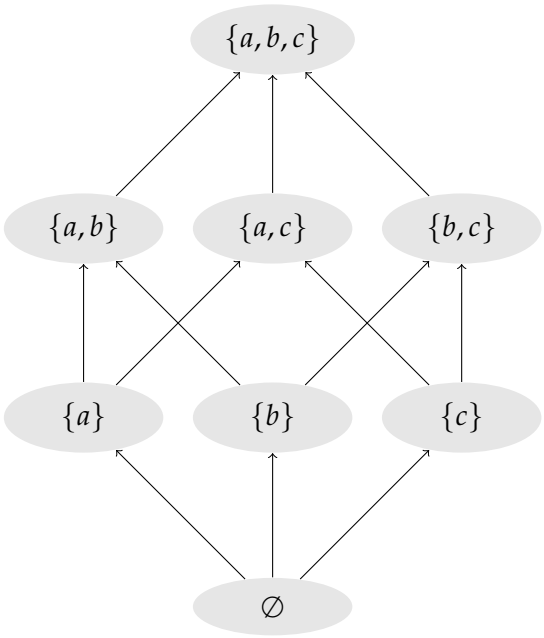

Figure 5: The elements of the power set of the set $C = \{a, b, c\}$ ordered in respect to inclusion.

As stated before, the detection of authentication tokens becomes intractable when the number of cookies is too high. However, it is possible to significantly reduce the search space with a smarter visit of the power set graph.

In [Calzavara *et al.*, 2014] the optimizations of the search algorithm are based on the following observations:

Observation 1 (Minimality Condition)**.** *If $A \subseteq \mathcal{C}$ is an authentication token then all its supersets authenticates as well.*

Therefore, we can remove from the graph visit all the supersets of A. That is due to the minimality condition dictated by Definition 1.

Observation 2 (Anti-monotonicity Condition)**.** *If $A \subseteq \mathcal{C}$ does **not** authenticate S to B then the same happens with any subsets of A.*

This observation resembles the anti-monotonicity of the frequent itemsets and it basically suggests to explore the candidate itemsets from the smallest to the largest to reduce the search space.

These optimizations are effective *only* after the detection of at least one authentication token as the pruning can occur. Therefore the worst-case time complexity of the algorithm is still exponential in the number of cookies. Specifically, this happens when the entire set of cookies $\mathcal{C}$ is the only authentication token.

For example, let $C = \{a, b, c\}$ a set of cookies and suppose that $A = \{a, b\}$ is the unique authentication token. If we apply the algorithm in [Calzavara *et al.*, 2014] we will perform the following visit:

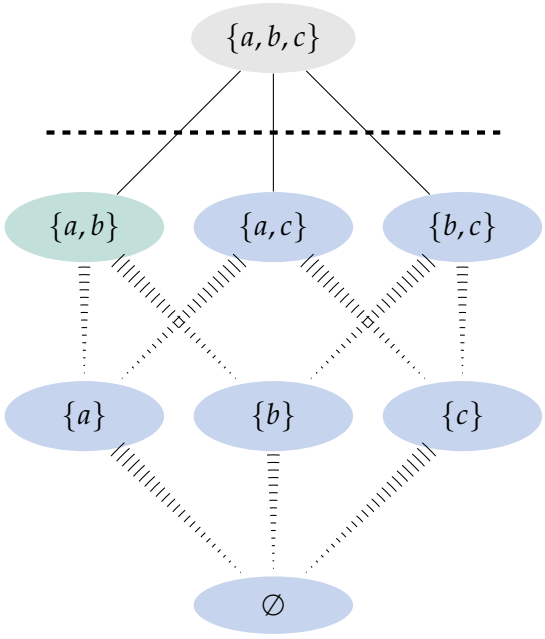

Figure 6: Smarter visit of the search space (best in colors). The green node correspond to the authentication token, the blue ones are the visited nodes while the gray one is the pruned one.

.

2.2.1 Proposed Solution

While we were investigating the authentication schemes in the most popular websites we notice that:

Observation 3. *Websites tend to have a unique authentication token or multiple overlapping ones.*

This observation lead us to the formulation of a **faster search algorithm**. The idea consists in detecting the **intersection** of authentication tokens. This intuition is better explained with an example.

Consider the previous example. Suppose we present to the server S the set of cookies $\{a, b, c\} \setminus \{a\}$ i.e. $\{b, c\}$. This set, by Definition 1, does not authenticate browser B to the server S. Therefore, we infer that the *cookie* $\{a\}$ *breaks the session* and it must be part of a unique authentication token or the intersection set of multiple ones. By excluding one cookie at a time we can single out the intersection set of the authentication tokens as stated in the following Lemma.

Lemma 1 (Intersection Lemma)**.** *Let* $\mathcal{C} = \{c_1, \dots c_n\}$ *be a set of cookies and let* $\mathcal{A} = \{A_1, \dots, A_m\}$ *be the (non-empty) set of authentication tokens included therein. Let:*

$$I = \{c_i \in \mathcal{C} \mid \mathcal{C} \setminus \{c_i\} \textit{ does not authenticate the client}\},$$

then $I = \bigcap_{i=1}^{m} A_i$.

Proof. To prove such a result, we must show that $I \subseteq \bigcap_{i=1}^{m} A_i$ and $\bigcap_{i=1}^{m} A_i \subseteq I$:

- Suppose that $c \in I$, we show that c belongs to every A_i, hence $c \in \bigcap_{i=1}^{m} A_i$. Assume by contradiction that there exists A_j such that $c \notin A_j$. This implies $A_j \subseteq \mathcal{C} \setminus \{c\}$, hence we know that $\mathcal{C} \setminus \{c\}$ authenticates. But this implies that $c \notin I$, which is contradictory;
- Suppose that $c \in \bigcap_{i=1}^{m} A_i$, then we know that $c \in A_j$ for each j. This implies that $\mathcal{C} \setminus \{c\}$ does not allow to authenticate the client, hence $c \in I$.

□

Finding I takes linear time $\mathcal{O}(n)$ where n is the number of cookies. From Lemma 1 we state the following corollaries:

Corollary 2. *If I authenticates B into server S then I is an authentication token for S and it is* unique.

Corollary 3. *If $I = \emptyset$ then the server S admits multiple non-overlapping authentication tokens.*

These achievements allows us to exploit the Observation 3:

- we are now able to detect an authentication token in **linear time** assuming such token is unique;
- we can prune from $\mathcal{P}(C)$ all the sets that does not contain I.

Unfortunately, if $I = \emptyset$ the worst time complexity is still exponential. However, Observation 3 guarantees that our method is linear in the average case: our experiments indicates that out of 220 websites analyzed 176 contain a unique authentication token.

Detecting the intersection has linear time complexity.

The complete algorithm for detecting authentication tokens consists of two phases: the first one searches for the intersection set I, while the second one is a smart bottom-up visit in the powerset graph. As we said before, finding the intersection of authentication tokens takes linear time and can be computed by presenting at the server S all the cookies subsets of size $k-1$ obtained by excluding one cookie. For each subset the function `isAuthenticated` checks whether $\mathcal{C} \setminus c$ authenticates B at S. A trivial definition of such an algorithm, which we may call `buildIntersection`, is given as follows:

Algorithm 1: Building the intersection set

Input : A set of cookies $\mathcal{C} = \{c_1, \ldots, c_n\}$ sent by a server S to the browser B upon login

Output: The intersection set of the authentication tokens $\mathcal{A}$ for S

```
1  begin
2      I ⟵ ∅;
3      foreach c ∈ C do
4          buff ⟵ C \ {c};
5          if not isAuthenticated(B, S, buff) then
6              I ⟵ I ∪ {c};
7          end
8      end
9      return I
10 end
```

Thanks to Corollary 2, it is possible to further reduce the search space by removing from computation all the supersets of cookies that does not contain I. Additionally, the minimality property tells us that as soon as we find a token that authenticates, by enumerating the subsets from the smallest to the largest, then such subset is surely an authentication token to be returned.

We can now define the function `Gen&Prune` that generates a collection of candidate cookies $cand^{(k)}$ of size $k > 1$:

$$\texttt{Gen\&Prune}(k, \mathcal{C}, \mathcal{A}, I) = \{C^{(k)} \subseteq \mathcal{C} \mid \nexists A \in \mathcal{A} : I \subset C^{(k)}\},$$

where $|C^{(k)}| = k$ with $k > |I|$, $\mathcal{A}$ is the set of authentication tokens found so far, I is the intersection set resulting from Algorithm 1 and the final algorithm for detecting authentication tokens is defined as follows:

Algorithm 2: Detecting Authentication Tokens

Input : A set of cookies $\mathcal{C} = \{c_1, \ldots, c_n\}$ sent by a server S to the browser B upon login

Output: A set of authentication tokens $\mathcal{A}$ for S

```
1  begin
2      I ⟵ buildIntersection(B, S, C);
3      if isAuthenticated(B, S, I) then
4          return {I};
5      else
6          A ⟵ ∅;
7          k ⟵ |I| + 1;
8          cand^(k) ⟵ {I ∪ {c_i} | c_i ∈ C \ I};
9          while cand^(k) ≠ ∅ do
10             foreach C^(k) ∈ cand^(k) do
11                 if isAuthenticated(B, S, C^(k)) then
12                     A ⟵ A ∪ {C^(k)};
13                 end
14             end
15             k ⟵ k + 1;
16             cand^(k) ⟵ Gen&Prune(k, C, A, I);
17         end
18         return A;
19     end
20 end
```

To build the gold set of cookies for S, we can eventually derive the labeling function $\hat{f} : \mathcal{C} \to \{0, 1\}$, where $\hat{f}(c) = 1$ iff there exists $A_j \in \mathcal{A}$. We repeat such a process for each web server S of a given list of web servers $\mathcal{S}$ to produce the complete gold set $\mathcal{G}$:

$$\mathcal{G} = \bigcup_{s \in S} \bigcup_{c \in \mathcal{C}_S} (c, \hat{f}(c))$$

where $\mathcal{C}_S$ denotes the set of all the cookies that a server $s \in \mathcal{S}$ has registered to the browser.

2.3 IMPLEMENTATION AND ASSESSMENT

We implement the algorithms, introduced in the previous section, using Python with **Selenium**: a suite of tools that automates web browsers across many platforms [Selenium, 2014]. We remark that Selenium does not emulate a browser (like for instance the Mechanize library) but it *automates* browsers, simulating the navigation of a human user. Therefore, even if websites limit the access to web crawler or web robots, according to the *Robot Exclusion Standard* [Lawrence and Giles, 1999], Selenium will not be affected by these restrictions since it is not seen as a robot but as a human entity. Furthermore, a very attractive feature of this tool is that it fully support **Javascript**: the scripting language of the Web that nowadays is becoming more and more essential for the correct functioning of sites.

Selenium is a toolkit that automates browsers and supports Javascript.

The script, we develop, is able to automatically login to almost any website and apply the algorithms discussed in the previous section to spot the authentication tokens building up our gold set of cookies. Certainly, one must possess a personal account on the considered websites for which the registration is inherently manual. Concretely, our script takes as input the set of user's credentials (username, nickname, email, password) and the url/s of the website/s to be analyzed, for which the user has already signed up. It then navigates to the given website and attempts to locate the login form by scraping the page. We claim that an HTML form is a login form if it has exactly one password field (`type=password`) and exactly one text field for the the user's credentials. If such form is not found in the current page, the script will crawl the website clicking on links/buttons that may potentially lead to a login form. For example, links named "Login" or "Sign In" will surely bring to a login page. Finally, if the form is been located and the authentication succeeded, the Algorithm 2 is applied on the registered cookies and the results, including the detected authentication tokens, are stored into a `Sqlite` database.

A login form has exactly one password field and one text field.

For convenience, we chose to automates two different browsers: one is used in the authentication phase, while the other one is used for the authentication token detection phase . The first one is `Mozilla Firefox`: a popular graphical browser that allow us to actually see how the script is interacting with the Web page and it has remarkable extensions like `Firebug`, which permits to, easily, monitor the network traffic.

The script automates two different browsers.

Graphical browsers are not suitable when performing Web tests: tasks that stress the browser with several requests and page loadings. The work of detecting authentication tokens reside in this category. Therefore, we chose to adopt, for the detection part, `PhantomJS`: a headless Web browser with fast and native support for various Web standards: DOM handling, CSS selector, JSON, Canvas, and SVG.

Listing 2: Help message from `detect_tokens.py`

```
> python detect_tokens.py --help

What it does
------------
1) Authenticates into given url(s);
2) Collects cookies;
3) Detects authentication token(s);
4) Saves results into a SQlite database.

Usage example
-------------
> python detect_tokens.py -e=user@mail.com -u=username
        -n=nickname -p=password -t=0.5 -i=http://example.com


optional arguments:
  -h, --help         show this help message and exit
  -f FILENAME        the file's name containing a list of urls.
  -i URL             input url
  -e EMAIL           your email
  -u USERNAME        your username
  -n NICKNAME        your nickname
  -p PASSWORD        your password
  -d DATABASE        the database name to store results in
  -t THRESH          the authentication threshold
  -k MAXTOKENS       maximum number of authentication tokens
  --ignore-alarm     skip any alerts dialog
  --timeout TIMEOUT  maximum time to process a url

manual mode:
  --manual           switch to manual login
  -s TIMETOLOGIN     number of seconds that you have to login
```

Understanding if the user has actually logged in is crucial for the correct behavior of our program. Therefore, the check performed by `isAuthenticated`$(B, S, \mathcal{C})$ must be as accurate as possible. To achieve this goal we carefully look for typical differences between the Web page before and after being authenticated.

We finally discover that after the authentication a typical Web page has the following characteristics:

1. it does not contain any login elements: text, links or buttons named “Login”, “Sign In” etc;
2. it does not contain any signup elements: text, links or buttons named “Sign Up”,“Join” etc;;
3. it contains the username or the email or the nickname of the user in the page source;
4. it contains some logout elements: text, links or buttons named “Log Out”,“Sign Out” etc.

These test-cases are carried out by analyzing the HTML page source once the user authenticates. For instance, if we are looking for a login element we search for the words: “login”, “sign in”, etc. Furthermore, to make the search more accurate we distinguish interactive login element, like buttons or links, from static text. It becomes evident that a Web page is to be assigned an **authentication score** $\delta \in [0, 1]$ a measure that captures the certainty of being authenticated and each case listed before contributes to increment δ.

However, since every website present different design policies, we still are not given to know what is the minimum authentication score to distinguish an authentication success from a failure. To overcome this problem we give as input to our script a threshold τ such that the function `isAuthenticated`(B, S, C) is defined as follows:

$$\texttt{isAuthenticated}(B, S, C) = \begin{cases} \mathit{True}, & \text{if } \delta > \tau \\ \mathit{False} & \text{otherwise} \end{cases}$$

In practice, our script turns out to be fast and adequately reliable. Failures happen when the authentication procedure requires to enter a `captcha`. Luckily, since such a procedure is carried out in the `Firefox` browser, it is been possible to implement a “*manual mode*”, giving the user the task of authenticating into the website. Once the authentication succeed, the script take control and proceed with the analysis.

Listing 3 present a snippet of the output of our script while authenticating and detecting the authentication tokens of `google.com`.

Listing 3: Log from `detect_tokens.py` while processing `google.com`

```
main: Starting Firefox.
main: Starting PhantomJS.

## PROCESSING URL 1 of 1
main: https://google.com
main: Extracted domain: 'google'
authentication_score: No sign up element found.
is_authenticated: Probability: 0.111
authenticate: Log in with email.
fill_login_form: Searching a login form.
search_login_form: Clicking on 'Accedi'
handle_modal_dialog: No frame found!
fill_login_form: Filling login form.
authentication_score: Username string found.
authentication_score: Email string found.
authentication_score: Nickname string found.
authentication_score: Nickname element found.
authentication_score: Email element found
authentication_score: Logout element found.
authentication_score: No login element found.
authentication_score: No sign up element found.
is_authenticated: Probability: 0.889
main: Login successful!

main: 7 cookies collected. Detecting authentication tokens.

detect_authentication_tokens: https://www.google.it/
detect_authentication_tokens: Checking input cookies.
build_intersection: Deleting cookie (1 of 7): 'SID'
authentication_score: No sign up element found.
is_authenticated: Probability: 0.111
build_intersection: SESSION BROKEN
build_intersection: Deleting cookie (2 of 7): 'HSID'
authentication_score: No sign up element found.
is_authenticated: Probability: 0.111
build_intersection: SESSION BROKEN
build_intersection: Deleting cookie (3 of 7): 'SSID'
authentication_score: No sign up element found.
is_authenticated: Probability: 0.111
build_intersection: SESSION BROKEN

...

build_intersection: Deleting cookie (7 of 7): 'NID'
authentication_score: Username string found.
authentication_score: Email string found.
authentication_score: Nickname string found.
authentication_score: Nickname element found.
authentication_score: Email element found
authentication_score: Logout element found.
authentication_score: No login element found.
authentication_score: No sign up element found.
is_authenticated: Probability: 0.889

build_intersection: Intersection found: [u'SID', u'HSID', u'SSID']

detect_authentication_tokens: Checking if [u'SID', u'HSID', u'SSID']
    is the unique authentication token.

...

detect_authentication_tokens: Found unique authentication token:
        [u'SID', u'HSID', u'SSID']
```

2.4 DATA DESCRIPTION

We run our script over **220** most popular websites according to the `Alexa` ranking. We build our gold set by gathering and classifying **2546** cookies, including **342** authentication cookies. Having built a gold set of cookies, we can finally understand better how web authentication is really deployed in practice and how helpful is our notion of authentication token. An investigation of the gold set, confirms the observations we made when designing our detection algorithm (see Section 2.2). Figure 7 (a) highlights the fact that websites typically empploy a unique authentication token and rarely more than two (Observation 3). It is important to remark that whenever a website present a unique authentication token, our algorithm is able to detects it in linear time.

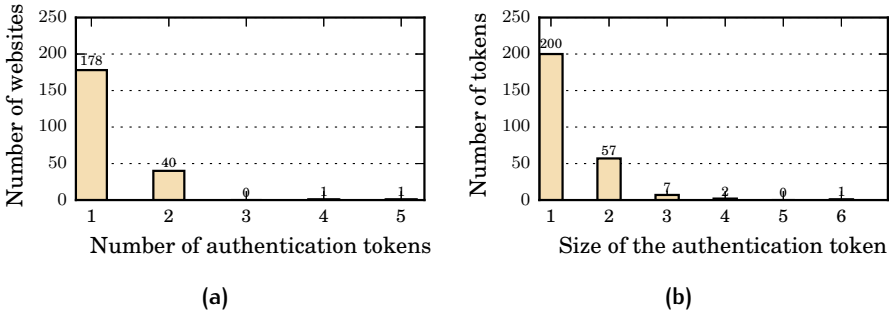

Figure 7: Distribution of the number of authentication tokens per website (a) and the number of cookies per authentication token (b)

Overall, we identified 271 authentication tokens distributed across the 220 websites: 203 tokens (74.9%) contain just one cookie, while 68 tokens (25.1%) are larger. Among these, 58 tokens are composed of two cookies: a manual investigation reveals that a fairly common practice for web authentication is to store the username in one cookie and some random session information in the other cookie. We summarize these numbers in Figure 7 (b) above. We also compute the number of overlapping authentication tokens i.e. tokens with cookies in common. Interestingly, this case is very rare: of all the 220 websites that we studied only 2 websites present such an authentication scheme. Finally, we conclude this section with two last remarks on our gold set. First, even though we automated the construction, clearly one still must possess a personal account on the considered websites to use our script: since the initial registration process is inherently manual, extending the gold set still requires some human effort. Second, we excluded from the gold set the known Google Analytics cookies since they are never used for authentication purposes and can be ignored by any authentication cookie detector employed by client-side defenses.

3 CLIENT-SIDE DEFENSES

In this chapter, we introduce and assess the quality of state-of-the-art authentication cookie detectors against our gold set. *Specificity*, *sensitivity* and *f-measure* are the validity measures we adopt to evaluate each client-side defense. We also introduce a novel measure called *protection score*, namely the percentage of protected websites according to Definition 2.
Our assessment turns out to be in agreement with the one performed by [Calzavara *et al.*, 2014]. Specifically, we conclude that existing authentication cookies detector are inaccurate leaving space for improvements in the detection process.

3.1 STATE-OF-THE-ART

In the field of client-side defenses against session hijacking we spot four existing authentication cookie detectors:

- SessionShield [Nikiforakis *et al.*, 2011] is a stand-alone proxy that interpose between the browser and the network protecting authentication cookies against XSS attacks by emulating the browser behavior implemented for the `Http-Only` flag;
- Serene [De Ryck *et al.*, 2012] is a client-side defense against session hijacking attacks. It is able to spot cookies which are likely used for authentication, but are not contained in any HTTP requests. Since these cookies can be easily stolen by a malicious script, they are cut away from HTTP requests and never used for authentication purposes.
- CookiExt [Bugliesi *et al.*, 2014] is an extension for the popular browser Google Chrome designed to ensure the confidentiality of authentication cookies against both XSS attacks and eavesdropping. CookiExt detect and applies both the `Http-Only` and the `Secure` flag to authentication cookies, while forcing a redirection from HTTP to HTTPS;
- Zan [Tang *et al.*, 2011] is an extension for the OP2 web browser aimed at protecting legacy web applications against different kinds of vulnerabilities. Notably, Zan tries to automatically apply the `Http-Only` flag to authentication cookies, to prevent their leakage via XSS.

Authentication cookies have notable discriminant features.

These tools requires first to detect the authentication cookies in order to apply some security policy. To achieve this, each of them use different heuristics based on a set of *hand-coded* rules, which rely on the same empirical observation: authentication cookies typically have longer and "more random" value than other cookies, and they often contain authentication-related words such as *"sess"*, *"auth"*, *"token"*, *"sid"* in their names.

Though the various tools differ in several aspects (for instance, randomness can be estimated in several ways, and different weights and thresholds can be chosen for the same feature), we abstract from the details here, and just summarize the aspects that are most relevant to our present needs. For any further information, we refer to the original papers.

Table 2 describes the checks performed by each detector. Cookies are denoted by pairs $c = (k, v)$ where k is the cookie's name and v is the cookie's value.

Table 2: Checks description performed to label the cookie $c(k, v)$ as an authentication cookie

Check	Description
$auth(k)$	k contains authentication-related terms
$known(k)$	k is a standard session identifier name
$len(v)$	the length of v is above a giben threshold
$H(v)$	the Shannon entropy [Shannon, 2001] of v is above a given threshold
$IC(v)$	the index of coincidence [Friedman, 1987] of v is above a given threshold
$s(v)$	the password strength [Florencio and Herley, 2007] of v is above a given threshold
$dict(v)$	v matches a dictionary word

Each detector can be represented as a boolean formula $\phi(c)$, which holds true if and only if c is recognized as an authentication cookie.

$$\textsc{SessionShield}(c) \triangleq (auth(k) \wedge len(v)) \vee (s(v) \vee \neg dict(v))$$
$$\textsc{Serene}(c) \triangleq known(k) \vee (auth(k) \wedge s(v) \wedge \neg dict(v) \wedge len(v))$$
$$\textsc{CookiExt}(c) \triangleq auth(k) \vee (IC(v) \wedge len(v))$$
$$\textsc{Zan}(c) \triangleq (auth(k) \wedge (H(v) \vee len(v))) \vee (\neg auth(k) \wedge H(v) \wedge len(v))$$

Table 3 highlights the checks performed by each cookie detector.

Table 3: Checks performed to label the cookie $c = (k, v)$ as an authentication cookie

Check	SessionShield	Serene	CookiExt	Zan
$auth(k)$	✓	✓	✓	✓
$known(k)$	✗	✓	✗	✗
$len(v)$	✓	✓	✓	✓
$H(c)$	✗	✗	✗	✓
$IC(v)$	✗	✗	✓	✗
$s(v)$	✓	✓	✗	✗
$dict(v)$	✓	✓	✗	✗

3.2 (RE-)EVALUATING EXISTING SOLUTIONS

3.2.1 Performance Measures

We implemented the four detectors in Table 2, and we fed them with all the cookies in our gold set, to then count the number of *true positives (tp)*, *true negatives (tn)*, *false positives (fp)*, and *false negatives (fn)* produced.

We refer to "positive" as any example in the gold set that is labeled as an authentication cookie, while we use the term "negative" for all the other examples. If the instance is positive and it is classified as positive, it is counted as a *true positive*; if it is classified as negative, it is counted as a *false negative*. If the instance is negative and it is classified as negative, it is counted as a *true negative*; if it is classified as positive, it is counted as a *false positive*.

A common way to represent the dispositions of the set of instances is through a two-by-two *confusion matrix* or contingency table [Fawcett, 2006]. This matrix forms the basis for many common metrics. In particular we compute two standard measures aimed at estimating the effectiveness of each detector:

$$specificity = \frac{tn}{tn + fp} \qquad sensitivity = \frac{tp}{tp + fn} \tag{1}$$

The *specificity* (also called *true negative rate*) measures the proportion of negatives which are correctly identified as such whereas the *sensitivity* (also called *true positive rate*) measures the proportion of actual positives which are correctly classified as such. We also consider an overall measure called the *f-measure* that is the harmonic mean of sensitivity and specificity.

$$fmeasure = 2 \times \frac{specificity \cdot sensitivity}{specificity + sensitivity} \tag{2}$$

Any authentication cookie detector having low *specificity* typically over-approximates the actual set of authentication cookies, that is, it makes many *fp* errors and may lead to usability problems, for instance by marking as `Http-Only` some cookies which should be legitimately accessed by JavaScript. Instead, any solution providing low sensitivity leans towards under-approximating the real set of authentication cookies, i.e., it makes many *fn* errors and leaves space for attacks. Clearly, not every *fp* will lead to a usability problem in practice and not every *fn* will correspond to a real security violation: understanding these aspects ultimately depends on the semantics of the specific client-side defense. However, the measures above do provide conservative estimation of the effects of deploying a new protection mechanism built over a given authentication cookie detector: as such, we believe it is very important to focus on them in the design phase of new defensive solutions.

3.2.2 Criticisms to Previous Assessments

The evaluation of existing authentication cookie detectors has so far been organized around (i) the collection of a dataset of cookies from existing websites, (ii) a manual investigation aimed at estimating the number of *fp* and *fn* produced by the detector. As stated in [Calzavara *et al.*, 2014] this approach suffers of two fundamental flaws:

1. First, and most importantly, the manual investigation is not supported by authenticating to the considered website, but rather by inspecting the structure of each cookie marked positive (or negative) by the detector: misclassifications are then estimated based on the expected syntactic structure of a standard session identifier. For example, any cookie containing a long random value which is negative by the detector is considered a *fn* (the inverse reasoning leads to an estimate of the number of *fp*). Clearly, this approach is convenient to perform in practice, but provides a very coarse and overly optimistic estimation, which is ultimately biased by the idea of assessing the effectiveness of the detector against the very same observations underlying its design.

2. The second flaw in existing assessment methods is that, in general, the number of *fp* and *fn* is not a statistically significant measure of the performance of a detector, since the distribution of the correctly labeled instances (*tp* and *tn*) cannot be ignored [Mitchell, 1997]. Unfortunately, the lack of any precise report about the number of *tp* and *tn* produced by previous detectors prevented us from computing specificity and sensitivity for existing evaluations, which is something we would have liked to consider for further comparison.

3.2.3 Results of Our Own Assessment

To evaluate the existing cookies detectors we fed to them our gold set of cookies and we construct for each one a *confusion matrix*, to which we compute the validity measures introduced previously.

Let us start with SessionShield whose confusion matrix is shown in the following table.

Table 4: Confusion matrix for SessionShield

		Predicted		
		Positive	*Negative*	Total
Actual	*Positive*	tp : 300	fn : 42	$tp + fn$: 342
	Negative	fp : 1264	tn : 940	$fp + tn$: 2204
	Total	$tp + fp$: 1564	$fn + tn$: 982	N : 2546

The sensitivity computed on SessionShield is impressive (88%) however the specificity is very low (43%) and this results in a poor 57% of f-measure. This coincide with the results obtained in [Calzavara *et al.*, 2014] and strongly conflicts the ones raised in [Nikiforakis *et al.*, 2011]. We move on with Serene whose confusion matrix is given in Table 5.

Table 5: Confusion matrix for Serene

		Predicted		
		Positive	*Negative*	Total
Actual	*Positive*	tp : 127	fn : 215	$tp + fn$: 342
	Negative	fp : 347	tn : 1857	$fp + tn$: 2204
	Total	$tp + fp$: 474	$fn + tn$: 2072	N : 2546

In opposition to SessionShield this detector present a high specificity 84% and a really low sensitivity 37% that lead to a 52% of f-measure. Next comes CookiExt whose confusion table is given in the following table:

Table 6: Confusion matrix for CookiExt

		Predicted		
		Positive	*Negative*	Total
Actual	*Positive*	tp : 193	fn : 149	$tp + fn$: 342
	Negative	fp : 506	tn : 1698	$fp + tn$: 2204
	Total	$tp + fp$: 699	$fn + tn$: 1847	N : 2546

This detector present quite more balanced validity measures: specificity 77%, sensitivity 56% and f-measure 65%. Still the number of true positive is particularly low.

Finally, we consider ZAN whose confusion matrix is given in the following table:

Table 7: Confusion matrix for ZAN

		Predicted		
		Positive	*Negative*	Total
Actual	*Positive*	tp : 191	fn : 151	$tp + fn$: 342
	Negative	fp : 452	tn : 1752	$fp + tn$: 2204
	Total	$tp + fp$: 643	$fn + tn$: 1903	N : 2546

The performance measures are similar to the ones in COOKIEXT with 79% specificity, 56% sensitivity and 66% f-measure.

In addition to these metrics, we measure the **protection** given by the detectors considered so far. According to Definition 2 a website is considered not vulnerable if and only if at least one cookie in each authentication token is identified and secured, using e.g. the `Http-Only` flag or the `Secure` flag. Formally, the protection given by a detector is defined as follows:

Definition 3 (Protection)**.** *Let W be the full set of websites where a detector D is tested. For a website $w \in W$, let $\mathcal{A}_w$ stand for the set of authentication tokens of w and let $tp_D(w)$ be the set of true positives produced by D on w. We define the* protection *granted by D on W, written $\rho_D(W)$, as follows:*

$$\rho_D(W) = \frac{|\{w \in W \mid \forall A \in \mathcal{A}_w : tp_D(w) \cap A \neq \emptyset\}|}{|W|} \tag{3}$$

We draw, in Figure 8, a bar chart that summarizes and compares the specificity, sensitivity and f-measure, together with the protection, of each detector.

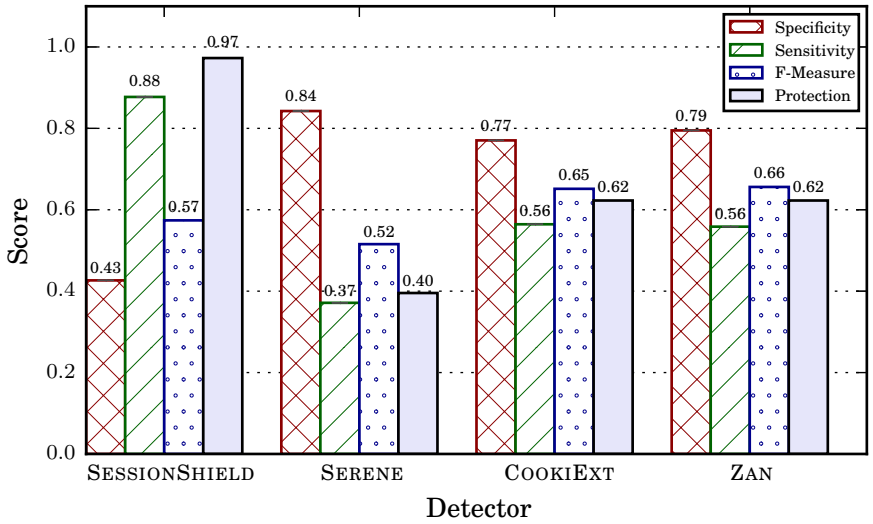

Figure 8: Performance evaluation in existing solutions

Our results are in agreement with the ones obtained by the authors of [Calzavara *et al.*, 2014]. Furthermore, we notice that the higher is the sensitivity, the higher is the protection. Specifically, from Figure 8, we see that the protection is higher with respect to sensitivity. This is due to the fact that some authentication tokens are made of multiple cookies and, thus, it is enough to identify at least one of them to shield the token.

It is, however, important to compare the protection with the other validity measures to properly assess the effectiveness of the detector. For example, SessionShield presents a satisfactory sensitivity (88%) and 97% of the websites are adequately protected, however the specificity is dramatically low (43%). Therefore, such detector is not fit for any practical client-side defense since it will lead to usability issues. In all the other cases, instead, the specificity is satisfactory but our evaluation highlights an unexpectedly low degree of protection.

4 A SUPERVISED LEARNING APPROACH

The previous chapter highlights the flaws and the inaccuracy of the existing client-side defenses. In this chapter, we demonstrate the effectiveness of machine learning by creating a novel authentication cookie detector, able to provide a high degree of protection while maintaining an adequate usability in the user's experience.

Creating such a detector is not trivial at all. The web is **highly heterogeneous**, each website has its own authentication scheme that may largely differ from the others. Consequently, authentication cookies detection becomes a difficult problem to solve due to the lack of a recurrent pattern that discriminate the authentication cookies from the other ones. This fact is the main reason why previous client-side defenses turned out to be imprecise and unreliable.

We show that far better results can be obtained by adopting a *supervised learning* approach in which a large set of N samples $\{x_1, \ldots, x_N\}$ called a *training set* is used to tune the parameters of an adaptive model. In this type of learning, the classes of the examples are known in advance and are represented with *target vector* y. Our authentication cookies detector is a *binary classifier* i.e. a function $y(x)$ that maps a cookie x to either a positive or negative class label. The precise form of the function $y(x)$ is determined during the training phase, also known as the *learning* phase, on the basis of the training data i.e. our gold set. Once the model is trained it can then detect authentication cookies, which are said to comprise a *test set*. The ability to categorize correctly new examples that differ from those used for training is known as *generalization*. For further details about this argument we refer to [Bishop *et al.*, 2006].

We study several popular classifiers and evaluate the performances of each one. Our assessments are based on a ROC analysis and on the validity measures previously introduced in Section 3.2.1 in order to make a comparison with the literature solutions.

4.1 FEATURE EXTRACTION

A fundamental step to design an effective supervised learning solution is to transform the original input variables into some new space of variables where, it is hoped, the pattern recognition problem will be easier to solve. This pre-processing stage is referred as *feature extraction*: the aim is to find useful features that are fast to compute, and yet also preserve useful discriminatory information enabling authentication cookies to be distinguished from non-authentication ones.

We compute, on our gold set of cookies, all the literature features introduced in Section 3.1 and also a novel class of *contextual* features proposed by [Calzavara *et al.*, 2014]. Subsequently, we draw the box plots [1], corresponding to numerical attributes, and the bar charts, corresponding to categorical attributes, to visualize the distributions of such features in order to understand their discriminant capacity.

Since we cannot rely just on visualization tools, we use a forest of trees to compute a scalar measure that captures the feature importances in classifying cookies.

4.1.1 Non-contextual Features

We first start by exploring the effectiveness of some features illustrated in Table 2, and exploited by the existing authentication cookie detectors. All these features are extracted from a single cookie $c = (k, v)$, without considering its "context", i.e., other cookies registered on the same domain. In the following list, we enumerate our most interesting findings about the contextual features.

- $known(k)$: checks if k is a standard session identifier name. Surprisingly, we notice that standard naming conventions for authentication cookie does not always involve that the cookie is actually used for authentication purposes. For instance, it is not true that cookies named `PHPSessid`, the standard name adopted by PHP to keep the user's session, are authentication ones. To conclude this, we matched the cookie names occurring in our gold set against an extensive list of 45 known standard names employed by SERENE . Figure 9 plots the distribution of the two classes of cookie with respect to $known(k)$. Interestingly, only 57 of the 112 cookies with standard session names are actually authentication cookies.

1 The box plot is a graphical tool that uses the median, the approximate quartiles, and the lowest and highest data points to convey the level, spread, and symmetry of a distribution of data values.

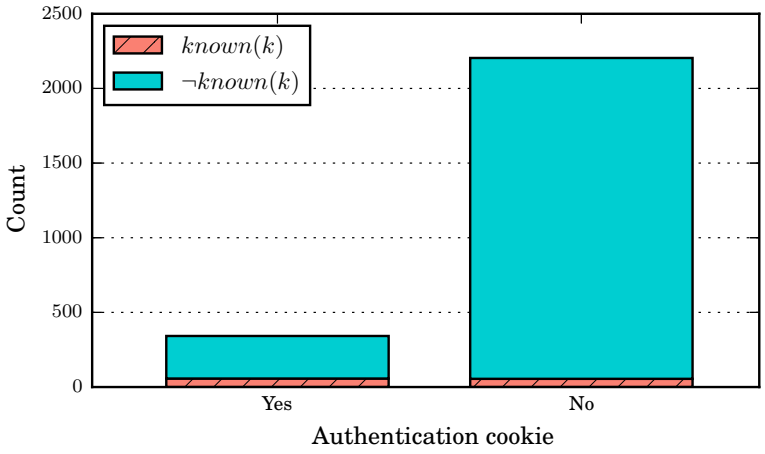

Figure 9: Distribution of the two class of cookies with respect standard naming conventions

There is no definite explanation about this unexpected behavior. Though we occasionally observed that web developers generate random cookies through some session management API of the underlying framework, and then populate other cookies with these randomly generated values to implement a custom authentication scheme.

- $auth(k)$: checks if k contains any authentication-related terms. A manual investigation of our gold set of cookies highlights some recurrent terms inside the authentication cookies names. Specifically, we isolate the words that seem to better distinguish authentication cookies: "sess", "login", "auth", "token", "acc", "key", "security", "token", "password" and "hash".

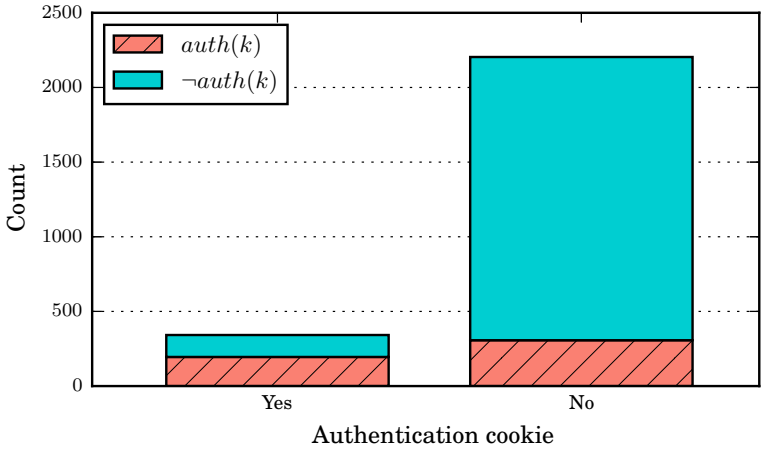

Figure 10: Distribution of the two class of cookies with respect to the presence of authentication-related terms

- $len(v)$: the number of characters in the cookie's value. The literature solutions, previously introduced, suggest that the length of a cookie value is useful to discriminate authentication cookies.

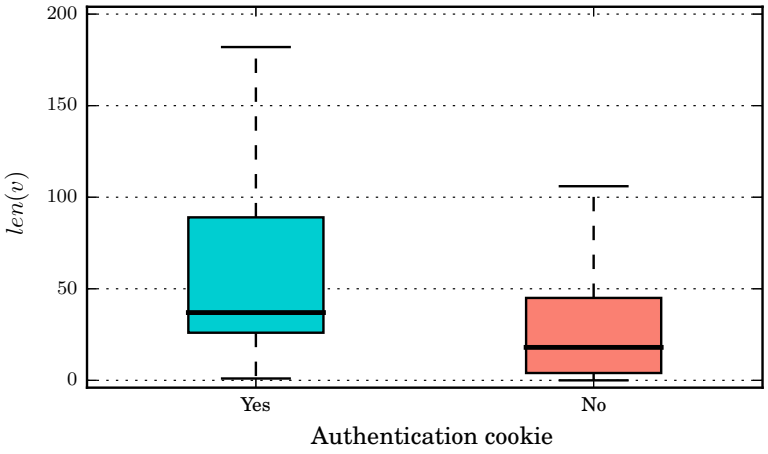

Figure 11: Distribution of the two class of cookies with respect to the length of cookie's value

We confirm that most of the authentication cookies are rather long as expected, in that their values include at least 25 characters, even though we observe that some authentication cookies are surprisingly short. A further investigation, reveals that such short cookies are generally combined with other authentication ones. We notice that some websites register a long cookie that identify the user's session together with a shorter one that may contain other user's information (email, username, etc).

- $s(v), H(v), IC(v)$: we notice that the value of authentication cookies is, typically, encrypted for security reasons. We know, from literature, that encrypted text is highly random and unpredictable. Therefore, we consider some randomness measures to provide additional discriminant power in our classifier. Specifically, we compute the Shannon entropy $H(v)$ and the index of coincidence $IC(v)$ of the cookie's value. We also consider the password's strength defined by the authors of [De Ryck *et al.*, 2012] as $s(v) = length(v) \cdot \log_2(alphabet(v))$, where $alphabet(v)$ is a score based on the occurrence in v of lower case letters, upper case letter, digits and special characters. We show the distribution of these measures in the following figure.

  We observe, from Figure 12, that authentication cookies generally have a "more random" value with respect to the other class. This fact is made very clear by the distribution of the index of coincidence $IC(c)$.

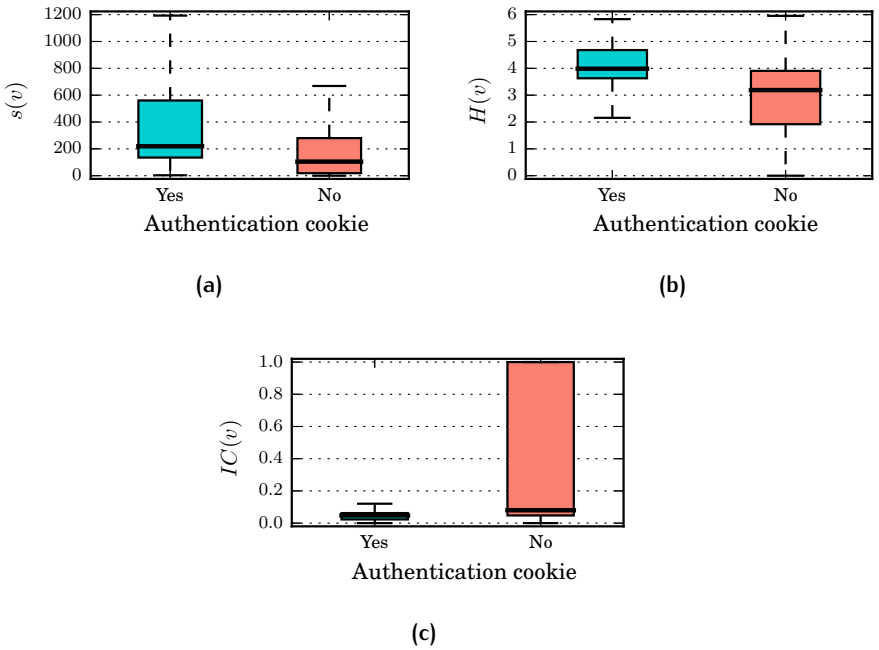

Figure 12: Distribution of the two class of cookies with respect to randomness measures.

- $js(c)$: checks if the cookie c is generated by Javascript. To perform such a control we explore the responses sent by the server in order to list the cookies set via header. By exclusion, the missing cookies are necessarily registered through a Javascript call. Before we started this analysis, we expect that authentication cookies should not be registered through Javascript since such cookies would become an easy target for XSS attacks. Our beliefs are confirmed as shown in the following plot.

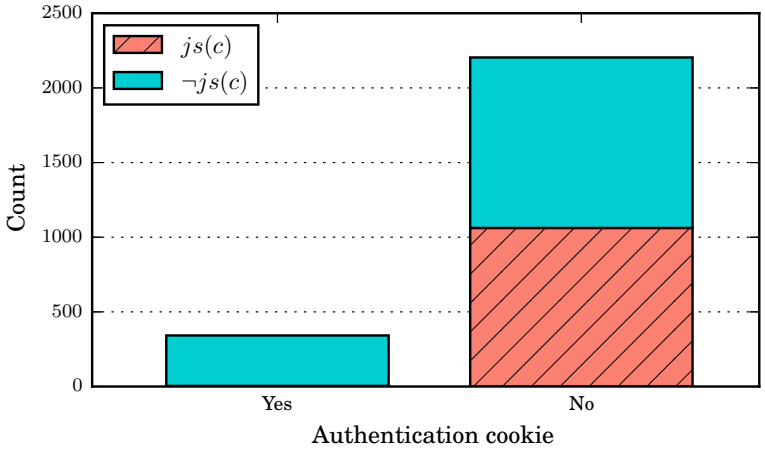

Figure 13: Distribution of the two class of cookies with respect to Javascript.

Our analysis suggests that the number of Javascript cookies used for authentication purpose is negligible: of the 220 websites we considered, only 4 of them use Javascript cookie for authentication. We carefully investigate these websites and we notice that they present more relaxed security requirements. We believe that these cases are justified from the fact that the user does not provide any particularly sensitive information that could make a session hijacking attack worth of. For instance, once we registered to `vevo.com`, (a popular website hosting music videos) we just unlock the possibility of saving a music playlist. Therefore, we think developers have focused on providing a more comfortable user experience by adopting Javascript in spite of weaker security measures.

- $expiry(c)$: a cookie can be assigned an expiration date by the remote server, to tell the browser to delete it after a given time. As depicted in Figure 14, the two class of cookies exhibit a similar distribution. However, although not so relevant, non-authentication cookies tend to have a longer expiration date. It would be legit to expect that authentication cookies typically had a rather short expiration date, since using the same cookie for a long period gives more time for session hijacking attack.

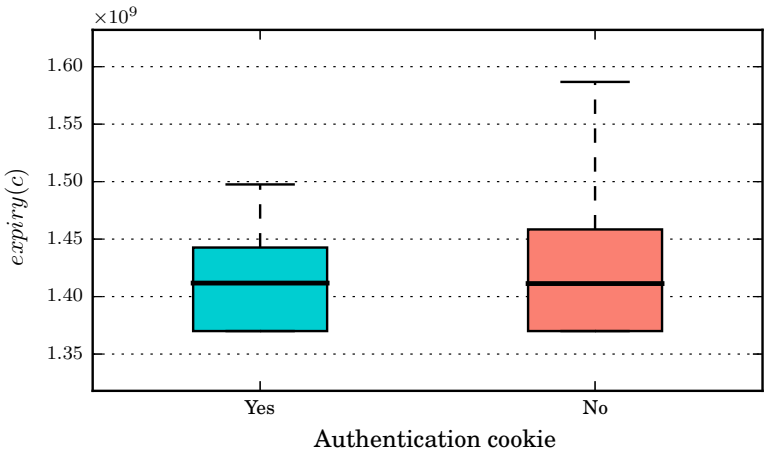

Figure 14: Distribution of the two class of cookies with respect to the expiration date (in Unix format)

By contrast, we noticed that several authentication cookies present a fairly late expiration date, as it is highlighted by the distribution plotted in Figure 14. It seems that different choices correspond to different web application semantics: security-critical websites (e.g., Dropbox) typically adopt authentication cookies with an early expiration date, while other websites (e.g. Twitter) prefer to enhance the user experience releasing authentication cookies which expire much later.

- $httponly(c), secure(c)$: cookies flagged as `Http-Only` or `Secure`. It would be legit to conclude that `Http-Only` and `Secure` cookies are likely used for authentication, since they are explicitly protected by web developers. Instead, several studies show that the adoption of these cookie flags is largely disregarded by existing websites ([Jackson and Barth, 2008], [Zhou and Evans, 2010]).

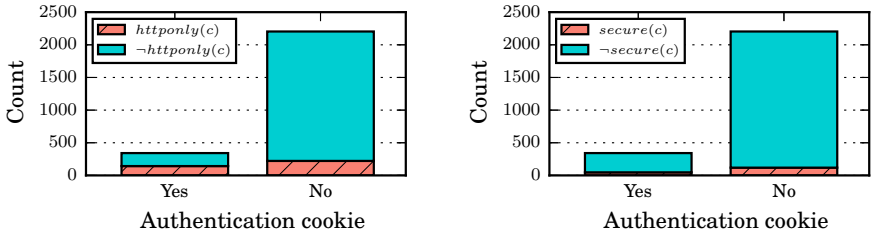

Figure 15: Distribution of the two class of cookies with respect to `Http-Only` and `Secure` flags

As depicted in Figure 15, the previous observation is confirmed: the flags do not provide a definite evidence of the cookie being used for authentication, although `Http-Only` is more adopted to protect authentication cookies with respect to the `Secure` flag.

To conclude, the following table summarizes the non-contextual features we explored so far.

Table 8: Features used to classify the cookie $c = (k, v)$

Name	Description
$known(k)$	k is a standard session identifier name;
$auth(k)$	k contains authentication-related terms;
$len(v)$	the number of characters of v;
$s(v)$	the password strength of v;
$H(v)$	the Shannon entropy of v;
$IC(v)$	the index of coincidence of v;
$js(c)$	c is a javascript-generated cookie;
$expiry(c)$	the expiration date of c;
$secure(c)$	c is flagged as `Secure`;
$httponly(c)$	c is flagged as `Http-Only`;

4.1.2 Novel contextual Features

After a careful evaluation of existing proposals, we manually inspected our gold set, seeking for novel distinctive features useful for detecting authentication cookies. The Web is highly *heterogenous*, meaning that some features are effective for a given website but not necessarily for another one. The authors of [Calzavara *et al.*, 2014] discovered, however, that novel discriminant features can be derived by considering the "context" of a given cookie c i.e. the set of all cookies $\mathcal{C}$ assigned by a given server.

- $tf\text{-}idf_{httponly}(c, \mathcal{C}), tf\text{-}idf_{secure}(c, \mathcal{C})$. We find out that the usage of the cookie flags `Http-Only` and `Secure` follows several different patterns, which should be effectively exploited to discriminate authentication cookies: first, there are websites which explicitly protect only cookies containing session information; then, we have some high-security websites, e.g., Dropbox, which protect all the cookies they register, irrespective of the nature of their contents; and finally, we have several websites which just completely ignore the adoption of the available cookie protection mechanisms.

  Intuitively, if a website registers a set of cookies, but only one (or very few) of those is labeled as `HTTP-Only`, then that cookie is likely used for authentication purposes. Instead, if all the cookies (or no cookies at all) from the website are labeled as `HTTP-Only`, the presence (or the absence) of the flag should not play any significant role during classification. Similar issues often arise in the area of information retrieval, where the effectiveness of a retrieval system depends also on accurately estimating the importance of words to each text document within its corpus. This idea is typically captured by the "term frequency-inverse document frequency" (tf-idf) score [Manning *et al.*, 2008], which increases proportionally to the number of times a word appears in a document, but is penalized by the frequency of the word in the whole corpus. Specifically, let $C = c_1, \ldots, c_n$ be a set of cookies registered by a given website and let n_h be the number of `HTTP-Only` cookies in $\mathcal{C}$. We define the "term frequency" of the `HTTP-Only` flag for a given cookie c_i as:

$$tf_{httponly}(c_i) = \begin{cases} 1, & \text{if } c_i \text{ is } \texttt{Http-Only} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

  We then define the "inverse document frequency" of the `HTTP-Only` flag with respect to the set $\mathcal{C}$ as:

$$idf_{httponly}(\mathcal{C}) = \log_2 \left(\frac{n}{n_h + 1} \right) \tag{5}$$

According to the definition of $tf\text{-}idf$, we then computer:

$$tf\text{-}idf_{httponly}(c_i, \mathcal{C}) = tf_{httonly}(c_i) \cdot idf_{httponly}(\mathcal{C}) \tag{6}$$

In Figure 1(e) we show how `HTTP-Only` cookies distribute over the measure above. The plot highlights that several authentication cookies appear whenever $tf\text{-}idf$ `HTTP-Only` is above a given positive threshold.

- $Z_{len}(c, \mathcal{C}), Z_H(c, \mathcal{C}), Z_s(c, \mathcal{C})$ We discovered before that authentication cookies typically contain rather long values. Still, two observations apply. First, the notion of "long" is inherently dependent on the specific website; indeed, it would be much more accurate to state that authentication cookies are usually longer than any other cookie registered by the same website. Second, we occasionally noticed some short authentication cookies that contradict the previous observation. However, we also discussed that these cookies are typically paired with other, longer authentication cookies, and protecting the latter would be enough to safeguard the website, as long as this allows for making all the authentication tokens invulnerable (Definition 2). Hence, reasoning at the website level seems effective also to protect scenarios implementing the authentication scheme discussed before, and we thus argue to consider the Z-score [Wylie, 1960] of the cookie length. In general, given a feature of interest, the Z-score measures the (signed) number of standard deviations an example is above the mean of the population. Concretely, for each $c_i = (k_i, v_i)$ in a set of cookies $\mathcal{C}$ registered by a given website, this is computed as:

$$Z_{len}(c_i, \mathcal{C}) = \frac{len(v_i) - \mu_{\mathcal{C}}}{\sigma_c} \tag{7}$$

where μ_c and σ_c are the mean and the standard deviation of the cookie lengths in $\mathcal{C}$, respectively. We show the distribution of the Z-score of the value length with respect to our two classes of cookies in Figure 1(f).

Table 9: Features used to classify the cookie $c = (k, v)$ within its context $\mathcal{C}$

Name	Description
$tf\text{-}idf_{httponly}(c, \mathcal{C})$	the $tf\text{-}idf$ of the `Http-Only` flag in c w.r.t $\mathcal{C}$
$tf\text{-}idf_{secure}(c, \mathcal{C})$	the $tf\text{-}idf$ of the `Secure` flag in c w.r.t $\mathcal{C}$
$Z_{len}(c, \mathcal{C})$	the Z-score of $length(v)$ w.r.t.$\mathcal{C}$
$Z_s(c, \mathcal{C})$	the Z-score of $s(v)$ w.r.t. $\mathcal{C}$
$Z_{IC}(c, \mathcal{C})$	the Z-score of $IC(v)$ w.r.t. $\mathcal{C}$

4.2 TRAINING A CLASSIFIER

In this work, we use Scikit-learn [Pedregosa *et al.*, 2011], a Python module integrating a wide range of state-of-the-art machine learning algorithms. We test several classification models evaluating their performances and comparing them with the literature detectors (see Chapter 3). In this document, we limit to report the four best classifiers according to our analysis. Specifically, we consider:

- AdaBoost: a boosting algorithm that works by repeatedly running a given weak learning algorithm on various distributions over the training data, and then combining the classifiers produced by the weak learner into a single composite classifier. [Freund, Schapire, *et al.*, 1996];
- ExtraTrees: a tree-based ensemble method that consists of randomizing strongly both attribute and cut-point choice while splitting a tree node. In the extreme case, it builds totally randomized trees whose structures are independent of the output values of the learning sample. The strength of the randomization can be tuned to problem specifics by the appropriate choice of a parameter [Geurts *et al.*, 2006];
- BernoulliNB: a Naive Bayes method for multivariate Bernoulli models (i.e. each feature is assumed to be a binary-valued Bernoulli variable) in which all attributes are independent given the value of the class variable [Zhang, 2004];
- Random Forest: a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest [Breiman, 2001];

4.2.1 Challenges and Methodology

In the previous section, we explore the features of our gold set of cookies, providing a more in depth understanding on how data is distributed and how relevant are the features we computed. Still, it is not given to know whether a machine learning approach is going to be effective. Ideally, a classifier should be able to find a tradeoff between bias and variance [Geman *et al.*, 1992]. Roughly speaking, this refers to the ability of a learning algorithm to flexibly fit other datasets than the one on which it is trained, while being able to perform effectively at least on the training se. Indeed, training an accurate classifier is a challenging task: in this section, we introduce the most relevant problems we encountered and how we deal with them.

The bias/variance tradeoff.

Features Importances

The high dimensionality of data poses challenges to learning tasks due to the *curse of dimensionality*. In the presence of many irrelevant features, learning models tend to overfit and become less comprehensible. In this contest, *features selection* is an essential step to develop successful data mining applications. It is the process of selecting a subset of original features, according to certain criteria [Guyon and Elisseeff, 2003]. The objective of feature selection is three-fold: speeding up a machine learning algorithm, improving the prediction performance, by defying the curse of dimensionality and providing a better understanding of the underlying process that generated the data.

In this work, we use a forest of trees to derive features importances. Each tree provides a relative measure known as "Gini importance" or "mean decrease impurity" and is defined as the improvement in the "Gini gain"[2] splitting criterion. By averaging those relative importances over several randomized trees we can reduce the variance of such an estimate and use it for feature selection [Archer and Kimes, 2008].

The following graph shows the features importances of our dataset evaluated using a forest of 250 trees. The bars are the feature importances of the forest, along with their inter-trees standard deviation.

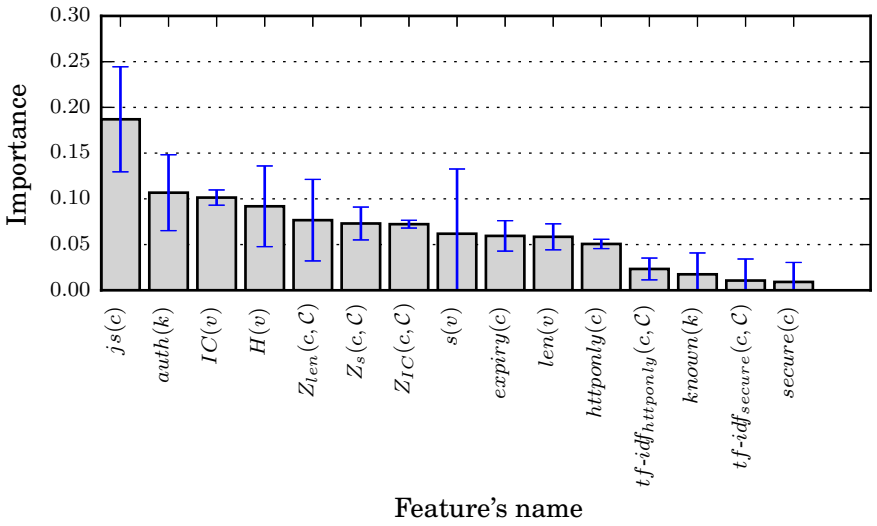

Figure 16: Feature importances using a forest of 250 trees

2 Also called "Gini index", measures the impurity of a split and is defined as follows:

$$g(t) = 1 - \sum_j p^2(j \mid t)$$

where $p(j \mid t)$ is the proportion of samples of class j in node t [Breiman *et al.*, 1984].

We extensively test our models by selecting different combination of features and we conclude that the best performances are achieved by using the **11 most relevant features**. Still, satisfactory results are achieved by selecting 8 features. We summarize all the features we considered, as long with their importances, in the following table.

Table 10: Features used to classify the cookie $c = (k, v)$

Name	Type	Contextual	Importances
$js(c)$	Boolean	✗	0.187
$auth(k)$	Boolean	✗	0.107
$IC(v)$	Numeric	✗	0.101
$H(v)$	Numeric	✗	0.092
$Z_{len}(c, \mathcal{C})$	Numeric	✓	0.077
$Z_s(c, \mathcal{C})$	Numeric	✓	0.073
$Z_{IC}(c, \mathcal{C})$	Numeric	✓	0.072
$s(v)$	Numeric	✗	0.062
$expiry(c)$	Numeric	✗	0.059
$len(v)$	Numeric	✗	0.058
$httponly(c)$	Boolean	✗	0.051
$tf\text{-}idf_{httponly}(c, \mathcal{C})$	Numeric	✓	0.023
$known(k)$	Boolean	✗	0.017
$tf\text{-}idf_{secure}(c, \mathcal{C})$	Numeric	✓	0.011
$secure(c)$	Boolean	✗	0.009

Overfitting

A common practice to avoid overfitting is cross-validation.

The problem of overfitting is one of the most widely concern when training a classifier. This phenomenon occurs when we fit the noise, present in our training data, into our learning algorithm rather than finding a general predictive rule [Hawkins, 2004]. To avoid overfitting, we adopt a procedure known as *stratified k-fold cross-validation*, whereby the original dataset $\mathcal{D}$ is randomly split into k subsets (the folds) $\mathcal{D}_1, \ldots, \mathcal{D}_k$ of approximately the same size. The model is trained and tested k-times; each time $t \in \{1, \ldots, k\}$, it is trained on $\mathcal{D}/\mathcal{D}_t$ and tested D_t. The cross-validation estimate of performance is then the average of the values computed in the loop. As suggested in [Kohavi *et al.*, 1995], we set $k = 10$ as the number of folds in our cross-validation procedure.

The Class Imbalance Problem

In our gold set of cookies, the class of non-authentication cookies is represented by a large number of instances while the other is represented only by a few. We estimate, approximately, one authentication cookie every 7 cookies. This fact may cause a significant bottleneck in the performance attainable by learning methods which assume a balanced class distribution.

This is known as *the class imbalance problem* and it is of crucial importance in machine learning, since it is encountered by a large number of domains [Japkowicz and Stephen, 2002].

A simple technique, to deal with the class imbalance problem, consist in artificially deleting (*undersampling*) or duplicating (*oversampling*) instances to balance the class distribution [Witten and Frank, 2005]. This methodology, however, has a pitfall: it physically changes the data, thus losing or altering information that may be useful to solve the classification problem. Consequently, we decide to adopt a **cost-sensitive** approach [Elkan, 2001]. The idea, here, is to assign a weight penalizing the misclassification error of the rarest class so as to rebalance the class distribution.

This reduces in defining a *cost matrix*, that provides the costs associated with each of the four possible outcomes shown in the confusion matrix which we denote as $\gamma_{tp}, \gamma_{fn}, \gamma_{fp}, \gamma_{tn}$. In the typical scenario, no costs are assigned to correct predictions, i.e., $\gamma_{tp} = \gamma_{tn} = 0$, while different weights may be assigned for the misclassification errors fn, fp. Specifically, if $\gamma_{fp} = \gamma_{fn}$ the misclassification costs are *uniform*. Instead, in order to rebalance the class distribution, we need to assign *non-uniform* misclassifications costs: the cost of misclassifying an instance of the rarest class is larger than the cost of misclassifying an instance of the most frequent class.

The `scikit-learn` tool provides a handy method that computes a cost matrix, rebalancing the class distribution. Let p and n be the number of positive and negative instances in the training set, respectively. We let:

Table 11: Cost matrix

		Predicted	
		Positive	*Negative*
Actual	*Positive*	$\gamma_{tp} : 0$	$\gamma_{fn} : \frac{\min(p,n)}{p}$
	Negative	$\gamma_{fp}: \frac{\min(p,n)}{n}$	$\gamma_{tn} : 0$

Since $p << n (342\ vs\ 2,204)$, it turns out that $\gamma_{fn} > \gamma_{fp}$, which means that our cost model penalizes more those errors occurring to instances of the positive class. The cost model is plugged into the learning process: this restricts the choice of the possible classifiers, since not every classifier supports cost-sensitive learning, but still we are able to identify good performing classifiers for our problem (see Section 4.3).

Hyper-parameters Optimization

Most learning algorithms expose a set of inputs, called *hyperparameters*, $\lambda \in \Lambda$, which change the way the learning algorithm itself works. For example, hyperparameters are used to describe a description-length penalty, the number of neurons in a hidden layer, the number of data points that a leaf in a decision tree must contain to be eligible for splitting, etc. The problem of identifying a good value for hyperparameters $\lambda \in \Lambda$ is called the problem of *hyper-parameter optimization*.The most widely adopted way of performing hyperparameter optimization is grid search, which is simply an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm. Since, this method is extremely expensive we focus on a faster alternative, known as **random search**, in which hyperparameters are randomly sampled from a parameter space with a specified distribution. Besides of the speedup, random search presents other interesting features that makes it more attractive with respect to grid search (see [Bergstra and Bengio, 2012]). We perform this analysis on our classification models by adopting a 10 fold stratified cross-validation and by using the f-measure as the score function to be optimized.

Randomized search versus grid search.

4.2.2 ROC Analysis

To assess the performance of our classifiers, besides the validity measures, we carry out a ROC analysis on each of them. Receiver operating characteristics (ROC) graphs are useful for organizing and selecting classifiers based on their performance. We are talking of two-dimensional graphs in which *tp* rate is plotted on the Y axis and *fp* rate is plotted on the X axis. The diagonal line $y = x$ represents the strategy of randomly guessing a class. The curves are traced by changing a threshold over the class probabilities predicted by the model: if the classifier output is above the threshold, the classifier returns 0, otherwise 1. Intuitively, we may imagine varying a threshold from $-\infty$ to $+\infty$ and tracing a curve through ROC space. Informally, one point in ROC space is better than another if it is to the northwest (*tp* rate is higher, *fp* rate is lower, or both) of the first.

ROC curves present an attractive property: they are insensitive to changes in label distribution. If the proportion of positive to negative samples changes, the ROC curves will not change. This fact is very useful in our environment: our dataset present a large class skew. Consequently, ROC graphs provide us a richer and more realistic measure of classification performance with respect to other evaluation techniques such as precision-recall graphs and lift curves.

For each classifier we perform a 10-fold stratified cross validation and plot the corresponding ROC curves. Let us start with the AdaBoost classifier, whose related graph is presented in the following figure.

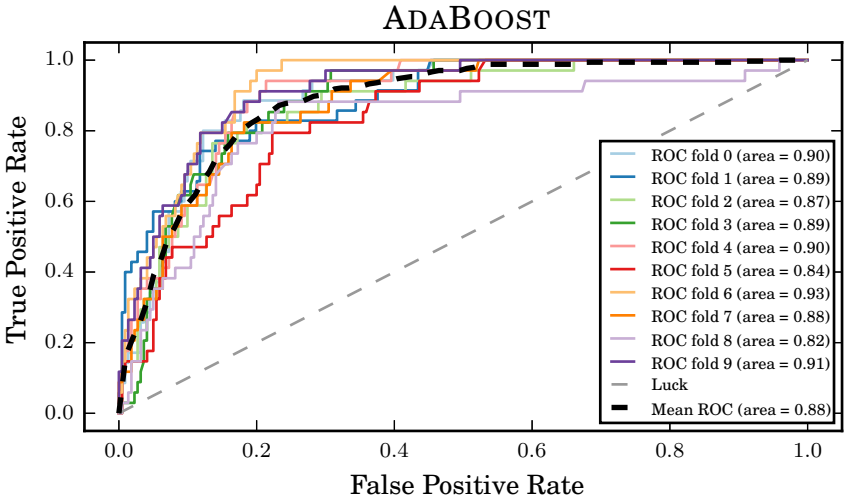

Figure 17: ROC graph for AdaBoost using 10-fold cross-validation.

A common scalar measure to summarize the performance of a classifier is the **area under the ROC curve** abbreviated *AUC* [Bradley, 1997]. The AUC of a classifier is the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance. Henceforth, we complete our ROC analysis by reporting all the ROC curves computed over the remaining classifiers.

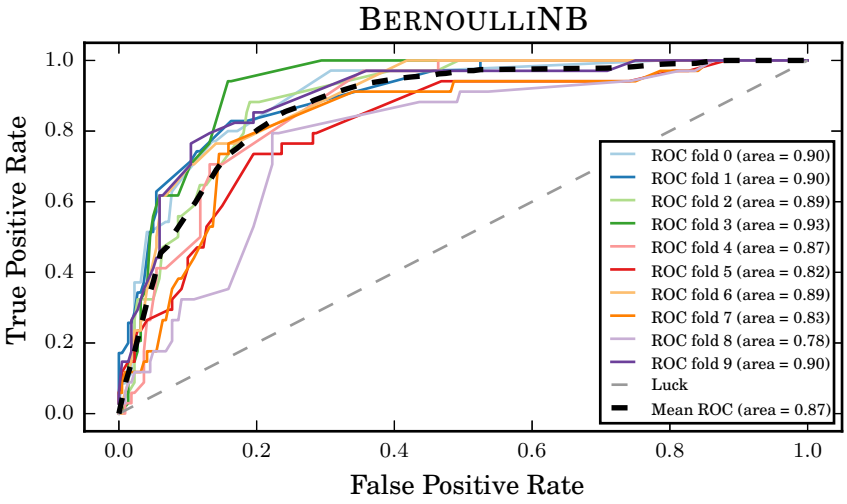

Figure 18: ROC graph for BernoulliNB using 10-fold cross-validation.

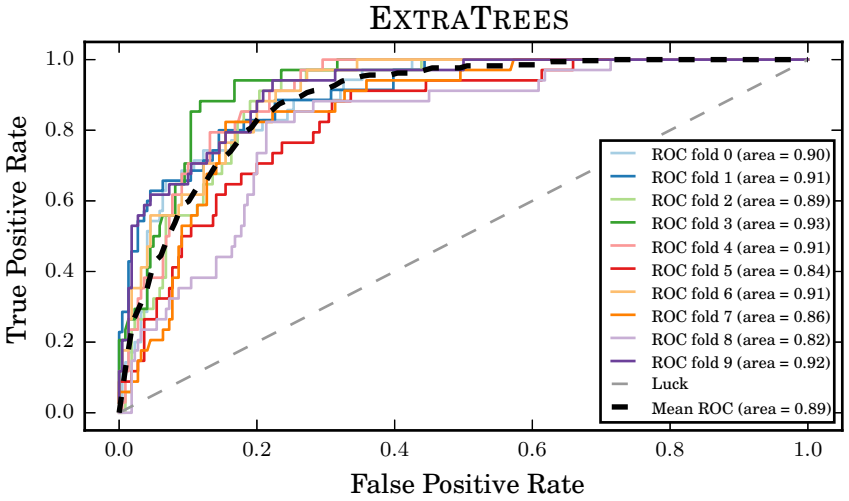

Figure 19: ROC graph for ExtraTrees using 10-fold cross-validation.

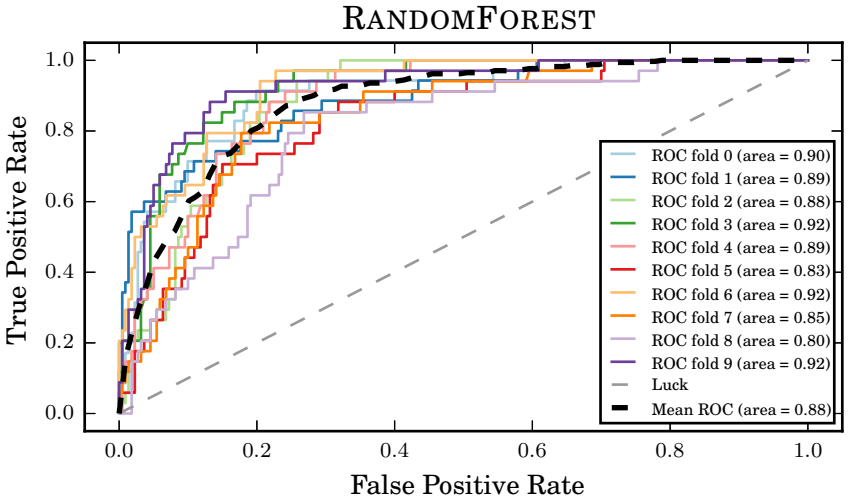

Figure 20: ROC graph for RandomForest using 10-fold cross-validation.

As depicted in the previous ROC graphs, we conclude that all the considered classifiers present a great mean AUC (> 0.8) and, therefore, a good classification performance *independently* from the class skewness in our gold set.

4.3 EXPERIMENTAL RESULTS

In this final section, we assess the performance achieved by the four classifiers. The measures of validity used to assess the performances of the classifiers are those we introduced in Section 3.2, i.e., *specificity, sensitivity, F-measure* and *protection*.

To understand the impact of the class imbalance problem (see Section 4.2.1), we, firstly, train our classifiers assuming a uniform class distribution. We summarize the experimental results in Figure 21.

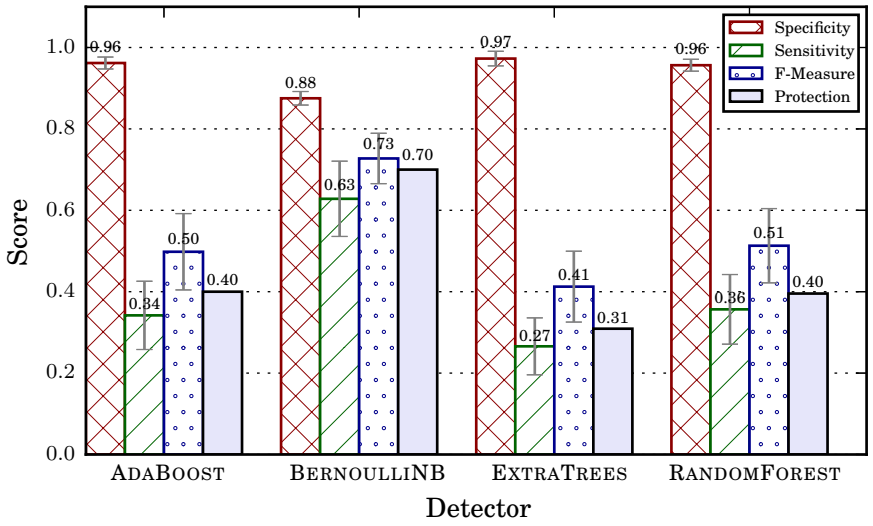

Figure 21: Performance evaluation of the best classifiers, assuming a uniform class distribution

It appears that the class skewness has a very negative impact on the classifiers performances. This contradicts the assessments of the ROC analysis. Therefore, we infer that the problem can be solved if we balance the class distribution. As we mention before, we do this by adopting a cost-sensitive approach, plugging into the training process the cost matrix defined in Table 11. The new performances achieved with this methodology are summarized in the following figure.

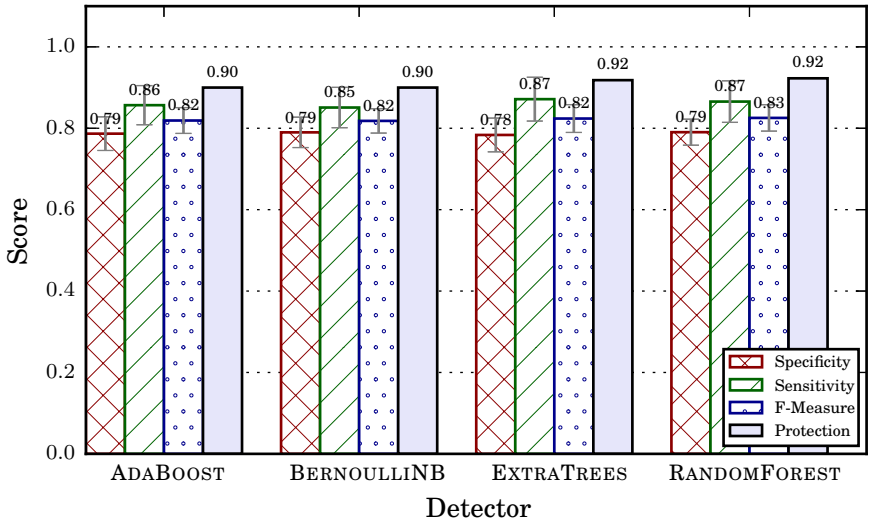

Figure 22: Performance evaluation of the best classifiers using a cost-sensitive approach

We immediately notice that adopting a cost-sensitive approach provides a large improvement and, most importantly, that our machine learning approach outperforms the discussed hand-coded heuristics, obtaining a remarkable trade-off between sensitivity and specificity. In that the F-measure of the worst classifier (0.82) is significantly higher than the F-measure of the best heuristic (0.66). Overall, we observe that the considered classifiers are able to protect around the 90% of the considered websites, while the best performing heuristics could only safeguard approximately the 60% of the websites, with the exception of the detector implemented in SessionShield , which however has a poor specificity and is not usable in practice. This huge improvement in protection does not come at the cost of usability, since the specificity of each classifier is no worse than the specificity computed for the different heuristics. Only the heuristic adopted in Serene is slightly better in this respect, but it could protect less than the 40% of the websites.

For sake of completeness, we report the two confusion matrices of the overall best performing classifier: RandomForest. Specifically, the first confusion matrix is drawn assuming a uniform class distribution.

Table 12: Confusion matrix for RandomForest (uniform)

		Predicted		
		Positive	*Negative*	Total
Actual	*Positive*	tp : 122	fn : 220	$tp + fn$: 342
	Negative	fp : 96	tn : 2108	$fp + tn$: 2204
	Total	$tp + fp$: 218	$fn + tn$: 2328	N : 2546

Instead, the second one, depicted in Table 13, is achieved by adopting a cost-sensitive approach.

Table 13: Confusion matrix for RandomForest (cost-sensitive)

		Predicted		
		Positive	*Negative*	Total
Actual	*Positive*	tp : 296	fn : 46	$tp + fn$: 342
	Negative	fp : 462	tn : 1742	$fp + tn$: 2204
	Total	$tp + fp$: 758	$fn + tn$: 1788	N : 2546

It is important to notice that manipulating the cost matrix, allows us to affect the trade-offs between security and usability: if security is the main concern, we can further increase the penalization cost of misclassifying an authentication cookie, thus leading to a higher sensitivity and, consequently, a higher protection.

5 CONCLUSIONS

In this thesis, we have shown that existing authentication cookie detectors perform much worse than expected when they are tested against a gold set of authentication cookies, confirming the results achieved in [Calzavara *et al.*, 2014]. Therefore, any browser-based defense built on top of them is bound to either negatively affect the user experience or provide an unsatisfactory level of protection. We then advocated the adoption of supervised learning techniques for the detection task and we showed a significant improvement with respect to state-of-the-art solutions, eventually striking a good balance between security and usability: compared to existing proposals, our solution provides a much stronger degree of protection, while being precise enough to be actually deployed in practice.

BIBLIOGRAPHY

Alexa, Team

2014 *The top 500 sites on the Web*, http://www.alexa.com/topsites. (Cited on p. 11.)

Archer, Kellie J and Ryan V Kimes

2008 "Empirical characterization of random forest variable importance measures", *Computational Statistics & Data Analysis*, 52, 4, pp. 2249-2260. (Cited on p. 43.)

Bergstra, James and Yoshua Bengio

2012 "Random search for hyper-parameter optimization", *The Journal of Machine Learning Research*, 13, 1, pp. 281-305. (Cited on p. 46.)

Bishop, Christopher M *et al.*

2006 *Pattern recognition and machine learning*, vol. 1, springer New York. (Cited on p. 33.)

Bradley, Andrew P

1997 "The use of the area under the ROC curve in the evaluation of machine learning algorithms", *Pattern recognition*, 30, 7, pp. 1145-1159. (Cited on p. 47.)

Breiman, Leo

2001 "Random forests", *Machine learning*, 45, 1, pp. 5-32. (Cited on p. 42.)

Breiman, Leo, Jerome Friedman, Charles J Stone, and Richard A Olshen

1984 *Classification and regression trees*, CRC press. (Cited on p. 43.)

Bugliesi, Michele, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan

2014 "Automatic and robust client-side protection for cookie-based sessions", in *Engineering Secure Software and Systems*, Springer, pp. 161-178. (Cited on pp. 6, 25.)

Callegati, Franco, Walter Cerroni, and Marco Ramilli

2009 "Man-in-the-Middle Attack to the HTTPS Protocol", *IEEE Security and Privacy*, 7, 1, pp. 78-81. (Cited on p. 6.)

Calzavara, S., G. Tolomei, M. Bugliesi, and S. Orlando

2014 "Quite a Mess in My Cookie Jar!: Leveraging Machine Learning to Protect Web Authentication", in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, International World Wide Web Conferences Steering Committee, Seoul, Korea, pp. 189-200, ISBN: 978-1-4503-2744-2, DOI: 10.1145/2566486.2568047, http://dx.doi.org/10.1145/2566486.2568047. (Cited on pp. v, xiv, 12, 15, 25, 28, 29, 31, 34, 40, 51.)

De Ryck, Philippe, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen

2012 "Serene: Self-reliant client-side protection against session fixation", in *Distributed Applications and Interoperable Systems*, Springer, pp. 59-72. (Cited on pp. 9, 25, 36.)

Elkan, Charles

2001 "The foundations of cost-sensitive learning", in *International joint conference on artificial intelligence*, vol. 17, 1, Citeseer, pp. 973-978. (Cited on p. 45.)

Fawcett, Tom

2006 "An introduction to ROC analysis", *Pattern recognition letters*, 27, 8, pp. 861-874. (Cited on p. 27.)

Fielding, Roy, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee

1999 *Hypertext transfer protocol–HTTP/1.1*. (Cited on p. 1.)

Florencio, Dinei and Cormac Herley

2007 "A large-scale study of web password habits", in *Proceedings of the 16th international conference on World Wide Web*, ACM, pp. 657-666. (Cited on p. 26.)

Franks, John, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart

1999 *HTTP authentication: Basic and digest access authentication*. (Cited on p. 5.)

Freund, Yoav, Robert E Schapire, *et al.*

1996 "Experiments with a new boosting algorithm", in *ICML*, vol. 96, pp. 148-156. (Cited on p. 42.)

Friedman, William F

1987 *The index of coincidence and its applications in cryptanalysis*, Aegean Park Press. (Cited on p. 26.)

Fu, Kevin, Emil Sit, Kendra Smith, and Nick Feamster

2001 *The Dos and Don'ts of Client Authentication on the Web.* (Cited on p. 6.)

Geman, Stuart, Elie Bienenstock, and René Doursat
1992 "Neural networks and the bias/variance dilemma", *Neural computation*, 4, 1, pp. 1-58. (Cited on p. 42.)

Genuer, Robin, Jean-Michel Poggi, and Christine Tuleau-Malot
2010 "Variable Selection Using Random Forests", *Pattern Recogn. Lett.*, 31, 14 (Oct. 2010), pp. 2225-2236, ISSN: 0167-8655, DOI: 10.1016/j.patrec.2010.03.014, http://dx.doi.org/10.1016/j.patrec.2010.03.014.

Geurts, Pierre, Damien Ernst, and Louis Wehenkel
2006 "Extremely randomized trees", *Machine learning*, 63, 1, pp. 3-42. (Cited on p. 42.)

Grossman, Jeremiah
2007 *XSS Attacks: Cross-site scripting exploits and defense*, Syngress. (Cited on p. 7.)

Guyon, Isabelle and André Elisseeff
2003 "An introduction to variable and feature selection", *The Journal of Machine Learning Research*, 3, pp. 1157-1182. (Cited on p. 43.)

Hastie, Trevor, Robert Tibshirani, Jerome Friedman, T Hastie, J Friedman, and R Tibshirani
2009 *The elements of statistical learning*, vol. 2, 1, Springer.

Hawkins, Douglas M
2004 "The problem of overfitting", *Journal of chemical information and computer sciences*, 44, 1, pp. 1-12. (Cited on p. 44.)

Hunter, J. D.
2007 "Matplotlib: A 2D graphics environment", *Computing In Science & Engineering*, 9, 3, pp. 90-95.

Jackson, Collin and Adam Barth
2008 "Forcehttps: protecting high-security web sites from network attacks", in *Proceedings of the 17th international conference on World Wide Web*, ACM, pp. 525-534. (Cited on p. 39.)

Japkowicz, Nathalie and Shaju Stephen
2002 "The class imbalance problem: A systematic study", *Intelligent data analysis*, 6, 5, pp. 429-449. (Cited on p. 45.)

Johns, Martin
2006 "SessionSafe: Implementing XSS immune session handling", in *Computer Security–ESORICS 2006*, Springer, pp. 444-460.

Kohavi, Ron *et al.*
1995 "A study of cross-validation and bootstrap for accuracy estimation and model selection", in *IJCAI*, vol. 14, 2, pp. 1137-1145. (Cited on p. 44.)

Kolšek, Mitja

2002 "Session fixation vulnerability in web-based applications", *Acros Security*, p. 7. (Cited on p. 8.)

Kristol, David M

2001 "HTTP Cookies: Standards, privacy, and politics", *ACM Transactions on Internet Technology (TOIT)*, 1, 2, pp. 151-198. (Cited on p. 1.)

Lawrence, Steve and C Lee Giles

1999 "Accessibility of information on the web", *Nature*, 400, 6740, pp. 107-107. (Cited on p. 19.)

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze

2008 *Introduction to Information Retrieval*, Cambridge University Press, New York, NY, USA, ISBN: 0521865719, 9780521865715. (Cited on p. 40.)

Mitchell, Tom M

1997 *Machine learning. WCB*. (Cited on p. 28.)

Nikiforakis, Nick, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen

2011 "SessionShield: Lightweight protection against session hijacking", in *Engineering Secure Software and Systems*, Springer, pp. 87-100. (Cited on pp. 25, 29.)

Owasp, Foundation

2013a *Http-Only*, `https://www.owasp.org/index.php/HttpOnly`. (Cited on p. 8.)

2013b *Session hijacking attack*, `https://www.owasp.org/index.php/Session_hijacking_attack`. (Cited on p. 6.)

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay

2011 "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, 12, pp. 2825-2830. (Cited on p. 42.)

Robertson, Stephen

2004 "Understanding inverse document frequency: on theoretical arguments for IDF", *Journal of documentation*, 60, 5, pp. 503-520.

Selenium

2014 *Selenium Project*, `http://docs.seleniumhq.org/`. (Cited on p. 19.)

Shannon, Claude Elwood

2001 "A mathematical theory of communication", *ACM SIGMOBILE Mobile Computing and Communications Review*, 5, 1, pp. 3-55. (Cited on p. 26.)

Tang, Shuo, Nathan Dautenhahn, and Samuel T King

2011 "Fortifying web-based applications automatically", in *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, pp. 615-626. (Cited on p. 25.)

Thomas, Stephen

2001 *Http Essentials with Cdrom*, John Wiley & Sons, Inc. (Cited on pp. 1, 3.)

Witten, Ian H and Eibe Frank

2005 *Data Mining: Practical machine learning tools and techniques*, Morgan Kaufmann. (Cited on p. 45.)

Wylie, Clarence Raymond

1960 "Advanced engineering mathematics". (Cited on p. 41.)

Zhang, Harry

2004 "The Optimality of Naive Bayes", in *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*, May 17-19, 2004, Miami Beach, Florida, USA, ed. by Valerie Barr and Zdravko Markov, AAAI Press. (Cited on p. 42.)

Zhou, Yuchen and David Evans

2010 "Why aren't HTTP-only cookies more widely deployed", *Proceedings of 4th Web*, 2. (Cited on pp. 13, 39.)

COLOPHON

NOTE: This document was typeset in LaTeX using `arsclassica`, a reworked version of the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".

Final Version: October 2014.