Analysis and Detection of Clickjacking on Facebook

Giulia Deiana

MEng Computer Science

Submission Date: 29th April 2015
Supervisor: Dr Gianluca Stringhini

Contents

Abstract

The rise and expansion of online social networks has created an enormous network of millions of people who constantly communicate and share content among each other. Attackers have quickly adapted and devised strategies to take advantage of unknowing users, mainly by luring them into their malicious domains to better execute their exploits. While most such attacks are well known and documented, not many studies have been done so far on the threat of clickjacking on social media. In this type of attacks, users are tricked into clicking a page element thinking to trigger a certain action, while unexpected ones happen instead. Such elements are commonly either hidden or "redressed" to make them look harmless and appealing; the consequences of a successful attack can vary greatly, as an attacker can make the user do anything requiring only clicks for execution, such as redirection to another malicious page, starting malicious downloads, or perform a Cross-Site Request Forgery attack if the user is currently logged in other services. The aim of this project is therefore to analyze the clickjacking problem on Facebook, to determine how common it is and the level of danger to which users are exposed if left unprotected. Using a web crawler, popular Facebook pages were analyzed and the comments from most recent posts retrieved. URLs found in comments were then collected and the HTML from the corresponding pages obtained in order to perform a static analysis for the presence of clickjacking. In order to do so, relevant features were selected and extracted from the raw data and 700 entries in the dataset were manually labeled as either clickjacking, malicious or benign pages. A supervised classifier was then trained on the tagged data and the resulting model applied to the remaining 8360 entries, successfully detecting a total of 71 clickjacking pages. The most common variants found were unwanted redirects to other malicious pages, likejacking, and promises of non-existent content in exchange for a click on a Facebook plugin. The findings confirm that the clickjacking problem is real and quite widespread and, even though not as dangerous as other types of attacks, still warrants the development of automated tools for its prevention and detection.

1. Introduction

Online social networks have revolutionized modern life in developed countries and have grown to become one of the most common ways for people to express their thoughts and ideas and communicate with their peers. Smartphones and other mobile devices have made it progressively easier to access the Internet at all times, making social networks all the more appealing and popular. Nowadays politicians and prominent figures use them to broadcast their ideas and gather more followers or their influence their voters. Companies advertise themselves and their products to the large social network audience. Knowledge of world problems is quickly shared among users to raise awareness and gather all possible financial and humanitarian help. News and facts, real or fake, spread quickly, shared in a chain by hundreds of people from all over the world. And hackers thrive in such an environment, where the large user pool allows them to easily find victims for their attacks. It is therefore increasingly important to perform in-depth studies of malicious activities in social media, as this will allow creation of automated tools that assist users in their daily navigation and make them far less likely to become easy preys.

1.1 Aims and goals

This project will focus on one particular social network, Facebook, with the aim of discovering to what extent its users are exposed to the threat of clickjacking, a malicious attack in which users are tricked into clicking selected elements in a web page to perform actions favorable to the attackers.

In order to do so, there are two main steps to be performed:
1. Gather enough data from malicious websites advertised on Facebook public pages.
2. Analyze it efficiently to detect clickjacking instances and separate malicious from benign activities.

The goal of this project is therefore twofold: deliver a web crawler that can collect significant amounts of relevant data from Facebook and the Internet, and analyze it to detect clickjacking and other malicious instances from it.

1.2 Implementation and results overview

The data that needed to be collected was in the form of full HTML code from suspicious pages found across Facebook comments. To collect this, the crawler's algorithm started with a few seed Facebook pages and used a BFS (breadth-first-search) approach in analyzing a fixed amount of posts and comments from every initial page before moving on to the next steps. All the comments found were saved to a text file in JSON format and URLs were extracted from those which contained links. Each URL found was then opened and its content saved in a text file if it contained at least one inline iframe, and a Google search for the comment itself was performed to determine if other Facebook pages contained it. The crawler then collected data from the first few Facebook pages returned by the Google search, effectively replacing the seed pages and thus starting a new cycle.

Around 1.7 GB of HTML pages have been collected. A close analysis of the data led to the identification of several features which mark important differences between genuine and malicious pages, while being at the same time able to identify clickjacking instances. Feature vectors were extracted from each entry to form a dataset matrix. A subset of the data was then manually labeled to form a training set over which a supervised classifier was trained. The model thus obtained was applied to the main bulk of the data to obtain predictions on whether each entry corresponded to benign, malicious or clickjacking pages.

The classifier identified 118 clickjacking instances out of 8360 data points, 71 of which were correctly predicted as clickjacking, while the remaining ones were false positives distributed into 15 benign and 32 malicious. Therefore the model was correct on 60.17% of its clickjacking predictions, making it fairly accurate when considering the small dataset over which it was trained.

1.3 Chapters summary

In the following chapter, context will be given to the reader, with a detailed review of literature found on the topic and its connection to the present document. Requirements will then be presented,

expanding on the introduction's aims and goals. The tools used to achieve the project's goals will also be introduced and discussed. Design and implementation details will then follow, with a detailed description of the algorithms developed and how the tools were used to implement them. The results of the analysis will then be covered and a series of case studies presented before concluding with observations on how the project could have been improved.

2. Context

In order to complete the project, sufficient background information about web crawling and clickjacking needed to be collected. Online sources and published papers were used for this purpose, and the result of this research is outlined in this chapter.

2.1 Why Facebook?

Created in 2004 by Mark Zuckerberg and other Harvard students, Facebook has grown to become the most popular social network in the world, with approximately 82.4% of users active outside of US and Canada [2, 4]. On December 2014 it had 890 million daily and 1.39 billion monthly active users [4]. A good 71% of Internet users are on Facebook [3] and the most vulnerable slice of the population, that is teenagers, uses predominantly Facebook as their social network of choice [1].
The company itself places great importance on security and has always been actively trying to reduce spam and malicious attacks on itself and its users. Facebook even warns and educates users about the most common attacks carried on though their website, as can be seen in their "Spam and Other Security Threats" page [5], which lists and describes the most common attacks and what users can do about them. Nonetheless, the very size of their social network makes perfect security a very ambitious goal and attackers still manage to find new victims and perpetrate their activities, mainly though clever social engineering.

2.2 Web crawling and Facebook

Gathering data from the web is no new activity. According to Wikipedia, "A Web crawler is an Internet bot that systematically browses the World Wide Web, typically for the purpose of Web indexing." [18]. Search engines and other sites routinely use web crawlers (also called "spiders") to find and index more pages and collect data.

Depending on their purpose, crawlers vary greatly in their function and implementations and can be extremely useful, but as they are able to gather far larger amounts of data in a short time compared to humans, they might affect deeply the performance and usability of servers. They consume a lot of bandwidth and may access a server too often, causing router and server crashes and thus impacting businesses negatively [18]. This is why policies and regulations have been established to control web crawling, and personal crawlers are somewhat frowned upon as they tend to try (knowingly or not) to evade them.

Social networks take crawling on their properties very seriously, as they already experience high load with normal human usage of their services without the aggravation of bots collecting their data. Facebook has their own, strict set of policies for crawling, which can be found in their robots.txt [20]. The document begins with:

"# Notice: Crawling Facebook is prohibited unless you have express written
permission. See: http://www.facebook.com/apps/site_scraping_tos_terms.php"

The link leads to their Automated Data Collection Terms, which contains a form that needs to be completed and sent to Facebook. If approved, a crawler compliant to their rules can be used in their domain. A form has been submitted to register the project's crawler, and an ethical approval from the UCL Research Ethics Committee (number 6521/002) obtained prior to start running the code. It is important to note that only public data has been collected during this project, and will only be used for UCL research purposes.

2.3 Background on Clickjacking

While browsing Facebook, a user reads a catchy comment promoting an interesting video, with a link provided. Intrigued, she clicks on it. Once loaded, the page looks fairly normal, with the promised video right in the middle of the screen, its content just a click away. All the user needs to do is press the play button. She does not know that over the innocuous looking "play button" an attacker placed an invisible iframe containing Facebook like buttons.

Image 1 – Likejacking example [11]

This is a classic example of clickjacking, where the attacker sets a bait for the user to click at and hides malicious content behind it, so that a user thinks to be clicking the bait element when in truth she clicked the invisible content. The name comes from the fact that the click is "hijacked", so that it goes to something which the user did not mean to click at all. Other names for clickjacking are "UI redress attack" and "UI redressing", which derive from the "redressing" of the clickable content by the attacker so to make it appealing to the unfortunate user.

Clickjacking attackers operate by modifying the content served from their own domain name and web servers and then tricking users into visiting their site to interact with its elements [15]. If the attack is

successful, a victim can be unknowingly lured into performing any action that can be done with a click, most commonly a redirect to another page, but sometimes also a download or an application execution. If she is logged in social networks or other platforms the attack variety widens to include Cross-Site Request Forgery, an attack that "forces an end user to execute unwanted actions on a web application in which they're currently authenticated" [13]. For example, on Facebook the user's account can be forced to post on one or more pages a spam message from the attacker, or could place a like on an unwanted page, ultimately spreading the threat and making other users more likely to fall on the same exact trap. The best freely available defense for the user at the moment is the Firefox browser addon No Script, which detects hidden elements in a page in real time and warns the user about the possibility of an attack [7, 10].

Clickjacking is not particularly widespread and the number of malicious pages fluctuates greatly. As can be seen from the Trend Micro data [12], there are considerable peaks around "fresh updates" on important world news or leakage of celebrity information/pictures that make it easier for attackers to construct successful baits. In the document, for example, the death of Steve Jobs corresponded to a large peak in number of clickjacking domains.

This document analyses the clickjacking problem from the user's point of view. The server's perspective is outside the scope of this document, but it should be noted that as of 2012 the large majority of websites did not take precautions against this threat. The main reasons for this was lack of awareness for the problem and not considering it a serious issue [14]. There are excellent documents and studies that explain how to implement good defenses that make clickjacking very hard to perform. Indications can be found in the "Clickjacking Defense Cheat Sheet" From OWASP [8] and in the Wikipedia page about Clickjacking [7], while a Stanford study on proper implementation of clickjacking server side defense can be found in "Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites" [9]. Finally, Huang et al. in "Clickjacking: Attacks and Defense" [15] developed new attacks, conducted the first user study on their effectiveness and built a tool to prevent them, encouraging browser and OS vendors to adopt such solutions in defending their users.

2.4 Clickjacking variants

In clickjacking attacks the user is tricked by an attacker into clicking on a target element that performs unintended actions. A very common vector for the target element itself is the popular HTML tag `<iframe>`. According to w3schools, "an inline frame is used to embed another document within the current HTML document" [16]. They can be styled with CSS and inserted inline and through scripts. `Iframes` are most commonly used to contain and display embedded content such as advertisements and videos, and in most cases their use is absolutely legitimate. Attackers, however, use them to embed content with malicious purposes and hide it where users will click it. Clickjacking comes in different flavors, the most common of which are described below.

In order to fool even more Internet-savvy people into clicking their malicious content, attackers need to hide the target element from the user's view. To achieve this, HTML and CSS styling features are used. Target elements can be made transparent by using "`opacity = 0`" or "`visibility: hidden`"

and placed on top of the bait. Alternatively, they can be placed underneath another element (which has been made unclickable) with a lower "`z-index`" value. Attackers can crop target elements by wrapping them inside `iframes` with negative offsets, so that only a part of them is visible; if inserted in the right context, the user won't think of it as malicious and might fall in the trap [15].

The most basic clickjacking attack consists in the placement of an `iframe` of `width=100%` and `height=100%` underneath the content of the malicious page, which only contain the target element made invisible by the attacker. This way, only the genuine page inserted in the `iframe` is visible. The target element placed in such a way that it overlaps with an element in the benign page that the user is likely to click, so that when this happens the target element is clicked instead.

The following variation uses JavaScript to cause a user to generate a click on the target element no matter where she clicks inside the malicious page. In this scenario, the attacker places the hidden target element in a hidden `iframe` and writes a script in JavaScript to allow it to follow the user's cursor wherever it moves. This way, when the user pauses to click something inside the page, she will actually be clicking on the target element which will be perfectly aligned under the cursor. In a similar fashion, the attacker can steal the user's click by anticipating it (in the case, for example, of a game) and move the target element underneath the mouse just before the click happens. In the same way, the attacker can ask the user to double click, and only place the target element under the mouse after the first click [15].

Some types of clickjacking have been nicknamed differently to reflect their characteristics. The particular case described in 2.3 and illustrated in Image 1 is a type better known as **Likejacking**. The target element is a transparent `iframe` containing the Facebook like plugin, which is placed over (or underneath) the bait element, which could be for example a play button or a "next" icon. If the user clicks on it while logged into Facebook with her account, the plugin will cause it to generate an automatic "like" to the attacker's referenced page without the user consent.

Another subtle type of clickjacking has been named **Cursorjacking**, as this time the user's cursor is hijacked to make it click in an unwanted location. The attacker makes the real cursor invisible and provides a fake one to the user, which can move it around just like a genuine one. The invisible real cursor is placed at a precise distance from the fake one. The attacker positions the target element at this exact same distance as the bait element, so that when the user clicks on the latter with the fake cursor, she is in fact clicking with her real one on the target element. The link provided in [17] offers a good interactive example of cursorjacking, from which an excerpt is in Image 2 below.

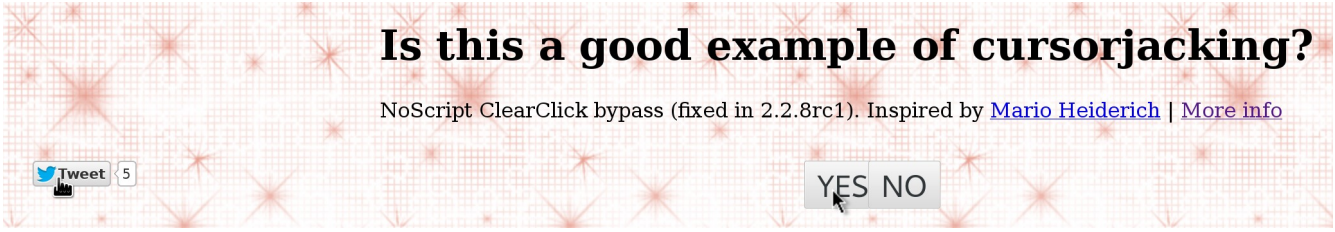


Image 2 - Cursorjacking [17]

The hand placed on the twitter button is the real cursor, which is enabled to allow users to understand the attack, but otherwise hidden in real cursorjacking instances. The cursor placed on the "YES" button is fake. The user thinks she is clicking on the "YES" button, when in reality only the twitter button is being clicked.

Other more common variants, however, are far simpler. Upon clicking on the target element, the user might:

- Be **redirected** to other sites and/or multiple windows containing malicious pages might be opened in her browser without consent.
- Start an automatic **download** of malicious material.
- Trigger the browser to open an external program to execute the attacker's code.
- Be convinced that the clicking action will correspond to a specific reward, which in fact will never be delivered (such as sharing on a social media in order to **unlock** a video, only to find that the video does not exist).

2.5 Related work

The following three papers are closely related to the core of the project. The discussion below outlines which parts of each were relevant, what was integrated into the implementation and in what ways their approach differed from what was used in this project.

2.5.1 "Crawling Facebook for Social Network Analysis Purposes" [19]

This paper describes two approaches to crawling the Facebook social network graph, Breadth-First-Search and Uniform sampling.

According to the paper, BFS is a very popular crawling approach for social networks and is particularly optimal and easy to implement for unweighted and undirected graphs. Seed nodes are provided from which the algorithm discovers their neighbors, placing them in a FIFO queue. Neighbors are then visited one by one in order of appearance, and the algorithm fills up the queue as it goes. As this process can virtually go on until the queue is emptied and there are no more nodes to visit, in practice a termination criteria is established in order to stop the crawler at a more reasonable time.

In Uniform sampling, the algorithm relies on the fact that the Facebook IDs are assigned by spreading them out in a 32-bit range space. Therefore, the crawling algorithm can generate random user IDs and test for their existence, in which case their page will be crawled along with the users in their friends list. The advantage of this approach is that the data will be far less biased than with BFS.

The paper then goes on to compare the datasets generated by each algorithm and produce metrics for which a discussion is outside of the scope of this project. This is because the data required here is of a different nature from what was collected by the paper authors. While their aim was to collect data from users and friends list, this project's focus on clickjacking makes it an essential requirement that

data from popular and/or dodgy public pages be collected instead. Most active users will notice if their account has been compromised and will try to clean it, thus removing evidence of the attack. Huge amounts of very fresh data would need to be collected to find clickjacking instances. On the other hand, pages with millions of visitors a day will find it extremely difficult to remove suspicious comments from their posts (admins of malicious pages will not even try, and will often post such things themselves), thus greatly increasing the chance of finding clickjacking links.

The full algorithm implemented for this project makes bias irrelevant and therefore the Uniform algorithm was not considered. However, the BFS algorithm described in this paper was of great use to this project, and in fact ended up being at the core of the final implementation, for which a full discussion will be provided in chapter 4.

2.5.2 "Prophiler: a fast filter for the large-scale detection of malicious web pages" [21]

This paper describes a method to perform computationally cheap and fast large-scale analysis of malicious web pages. They collected HTML pages and examined it statically for malicious content by extracting a number of features related to known attacks. In order to separate benign pages from malicious ones, they describe 77 different features, 19 of which from HTML content, 25 from JavaScript and 33 from the URL and host-based analysis. They produced a filter trained on a portion of labeled data using multiple machine learning classifiers and used a validation set to evaluate their model. The results were very good, as more than 85% of malicious pages were identified without the load and cost of a dynamic analysis.

The method described is absolutely relevant to the project, and it has been of great inspiration in choosing the approach to take in analyzing the collected data. It shows that good features can lead to a very good score by just performing a HTML static analysis. Because their were looking for malicious web pages in general, and not clickjacking ones, some features are not applicable. These ones however are particularly relevant in a clickjacking context:

- Number of `iframe` tags.
- Number of elements with a small area.
- Number of known malicious patterns (they used patterns established in drive-by-download attacks, but it can be adapted to clickjacking).
- Number of times a script contains "iframe".
- Number of suspicious URL patterns.
- Presence of a subdomain in URL.
- Presence of IP address in URL.
- Host Name (domain) registration date.

As most of these features are used in the project's code, they will be discussed in details in the implementation chapter.

2.5.3. "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code" [22]

This paper focuses on detecting drive-by-download attacks by performing a dynamic analysis of JavaScript code in a page. Their approach consisted in emulating the behavior of benign web pages and generating a number of features that clearly mark their non-maliciousness. When an unseen page is presented it is then emulated to check how far from the established profiles it is. An important thing to note is their effort in deobfuscation of scripts in malicious pages, which would surely be relevant for this project.

This work, though not as relevant as the one mentioned above, is still important as many clickjacking attempts would be detected with such an in-depth dynamic analysis. However, as clickjacking can and is often perpetrated with static elements, too many important features would be missed if such an approach would be used alone. After tweaking the features to be more clickjacking oriented, this excellent type of JavaScript analysis combined with the work in the above paper would surely detect successfully the vast majority of clickjacking attempts.

2.5.4 "A Solution for the Automated Detection of Clickjacking Attacks" [24]

While the previous papers focused on detecting malicious activity in web pages, the goal in this particular paper was to develop a much needed tool for automated detection of clickjacking attacks during normal browsing. The team delivered a browser plug-in called ClickIDS, presented an estimation of the prevalence of clickjacking attacks on the Internet and assessed whether proper defenses have been widely adopted.

Their clickjacking detection technique consisted in having a browser render the web page, then extract the coordinates of all clickable elements and programmatically control the cursor and keyboard to click on each of these elements. Their architecture consisted in a testing part responsible for the automated clicks and a detection part that detected probably clickjacking attempts. The latter unit was made of two separate browser plugins:

- ClickIDS, developed by the authors, which reports suspicious behavior when it detects that two or more clickable elements of different pages overlap at the coordinates of the mouse click.
- NoScript, which contains the anti-clickjacking feature ClearClick that allows it to trigger an alert when the mouse click is detected in a region (not at the exact coordinates as in the above) where elements from different origins overlap. The authors slightly modified its default output to make the evaluation easier.

The findings of this paper are very interesting and relevant, as they offer an idea of what to expect when searching for clickjacking attacks on the web. They ran their experiments for about two months, visiting a total of 1,065,482 unique web pages. Around 7% of them were unavailable (either down or under construction). From the remaining pages, 37.7% of them had at least one `iframe`, but only 930 pages contained a completely transparent `iframe` and 627 contained partially transparent `iframes`.

The results are outlined in the table below, which was taken from the results chapter (Table 1 in the paper).

	Value	Rate
Visited Pages	1,065,482	100%
Unreachable or Empty	86,799	8.15%
Valid Pages	978,683	91.85%
With IFRAMEs	368,963	37.70%
With FRAMES	32,296	3.30%
Transparent (I)FRAMEs	1,557	0.16%
Clickable Elements	143,701,194	146.83 el./page
Speed Performance	71 days	15,006 pages/day

Image 3 - Statistics of the visited pages [24]

Even more interesting are the results on the pages for which their detection tool generated an alert. The two plugins raised a total of 672 (137 for ClickIDS and 535 for NoScript) alerts, but the value drops to 6 alerts if one considers the case where both reported a clickjacking attack at the same time. The authors report that NoScript generated 97% of its alerts on pages containing no transparent elements at all. They then proceeded to manually analyze the output to determine which web pages generated false positives and which didn't. Image 4 (Table 2 in the paper) illustrates their findings.

	Total	True Positives	Borderlines	False Positives
ClickIDS	137	2	5	130
NoScript	535	2	31	502
Both	6	2	0	4

Image 4 - Results [24]

As can be clearly seen, only two out of the analyzed 978,683 valid pages were real clickjacking ones. The authors argue that most of the false positives were caused by either malicious pages that are not clickjacking or genuine elements that were just placed in a slightly unusual way. The borderline cases are very hard to classify, as they only fit in a more relaxed definition of the attack.

This paper shows how clickjacking pages are quite uncommon and difficult to find. The attack's most distinctive features generate many false positives that require human intervention to distinguish from real positive cases. This project differs from the paper's approach, as it does not do dynamic analysis of the crawled pages, but tries to use features derived from static HTML to determine if a page contains a clickjacking attack or not. Nonetheless, this paper's finding are sound and it is not to be excluded that a similar genuine/clickjacking-malicious ratio could actually reflect the reality of things and, therefore, this project's results as well.

2.6 Novelty of approach

Literature search suggests that no research has ever been done on analysis of clickjacking instances among malicious comments shared in the Facebook social media. The papers described above proposed solutions well suited to finding either malicious web pages in general or focused exclusively on detecting clickjacking attacks in real time. This project tries to understand whether applying one or more of these approaches to pages gathered from dodgy Facebook comments yields good results in finding more clickjacking instances, and is therefore novel material.

3. Requirements and tools

3.1 Problem statement and requirements

Being the most popular social network at the moment, Facebook is a huge attraction for cyber criminals, as the large pool of users makes it far more likely for them to find people who will fall for their scams. Because they have no direct control over Facebook pages themselves (the company tries to defend itself and its users as much as possible, so it leaves no vulnerabilities to be exploited), attackers are forced to be creative and always find alternative solutions to bypass security measures and perpetrate their activities.

The most common way for them to do so is to use advanced social engineering techniques to lure users into visiting their malicious domains. To achieve this, they may create seemingly harmless Facebook pages filled with links directed to their sites. More often, however, they create fake accounts to post catchy comments containing such links to popular public pages, so that they can obtain plenty of user visibility and increase the probability of fooling users. In short, their main goal in social medias is aggressive advertisements of their malicious web pages.

This project aims to collect a large number of such comments and analyze them to find instances of clickjacking attacks. Therefore, the problem statement can be summarized in the following questions: among all malicious domains advertised on Facebook, how many of them actually contain clickjacking attacks? And if yes, what types? Could they be easily blocked?

That the requirements for the project are as follows:

1. The implementation of a web crawler capable of gathering a large number of comments from Facebook pages that are most likely to contain malicious ones.
2. A tool that can collect the full HTML code of pages linked in dodgy comments.
3. The identification of features for detection of malicious pages and clickjacking instances among them.
4. The development of an algorithm capable of extracting such features from the HTML bulk and perform analysis to determine whether a page is benign, malicious or clickjacking.

3.2 Tools

3.2.1 Data collection tools

To interact with pages like a user would, a browser is needed. Therefore, in order to build the crawler the following options were selected: Google Chrome, Selenium + PhantomJS and PhantomJS alone. With the idea of performing dynamic analysis of collected web pages, the development of the crawler started with Selenium + PhantomJS.

Selenium is a tool for browser automation. It is mainly used for testing purposes, as it allows to easily create browser-based automated suites and tests, and offers support for all main browsers, including PhantomJS [24].

The best description for PhantomJS can be found on their website: "PhantomJS is a headless WebKit scriptable with a JavaScript API. It has fast and native support for various web standards: DOM handling, CSS selector, JSON, Canvas, and SVG." [25]. As PhantomJS uses WebKit, exactly like Google Chrome, the headless feature made it the browser of choice for the implementation of the crawler. Moreover, it is relatively straightforward to understand and its basics are well documented, though more advanced features lack proper disclosure and explanation (StackOverflow makes up for it, however). There are several variations of PhantomJS, the most notable of which is CasperJS, a wrapper that offers most features in a more convenient and accessible API.

There are two main reasons that led to using standalone PhantomJS as opposed to the Selenium + PhantomJS combination:

1. In order to discourage web crawling on their domains, Facebook makes it extremely complicated to extract data from their pages without using their API. After numerous failed attempts to collect the data by other means, the API approach was chosen as it allows to collect data in a much cleaner and faster way, not to mention it greatly reduces simplicity of the algorithm and code writing process. Therefore "clicking on the Facebook page like a user would" was no longer necessary, and given that Selenium automation helped mostly in that respect, the need to use it ceased to exist.

2. After accurate review of existing literature, static HTML analysis was chosen over a dynamic approach, thus reinforcing the idea that standalone PhantomJS would be more than enough for this project's purposes.

Note that, after this decision, Python could have been used instead of a browser. However, it seemed more appropriate to learn a new language (JavaScript) and experiment with headless browsers rather than always going for the "pythonic" approach.

3.2.2 Analysis and evaluation tools

In order to analyze the large amounts of HTML documents collected along with the corresponding

Facebook comments, Python scripts were used. In particular, Python NumPy was used to construct the feature set and Python scikit-learn for the machine learning part.
Alternatives are Matlab or Mathematica, but they are not free to use and largely unknown outside of academic environment. Java could have been an option, but Python was deemed to be cleaner and simpler to understand and use.

4. Design and implementation

This chapter gives a detailed description of the deliverables of the project. The web crawler and HTML extraction tool will be discussed in section 4.1, while the scripts used for analysis and evaluation are covered in 4.2.

4.1 Data gathering

This section describes the web crawler's algorithm and implementation. After giving an overview of the algorithm, the difficulties encountered in the development phase will be explored and discussed, before finally focusing on how the algorithm itself was actually implemented.

4.1.1 Algorithm overview

The input given to the crawling algorithm is a number of seed Facebook pages. Busy and popular pages are strongly preferred, as they are "liked" by more users attract a larger number of spammers and cybercriminals, who start disseminating their malicious comments in the page's posts in the hope that some user will fall in the trap and take their bait. At least 3 pages are recommended.

The algorithm's objective is to output the HTML of suspicious web pages found in Facebook comments. For each raw HTML text, the corresponding page's URL and the original comment from which the link was extracted are also given. The data is organized in a predefined format that allows the analysis and evaluation script to easily collect each page separately from the bulk.

The algorithm comprises five main steps:

1. Gather n posts from each seed page and put them in a FIFO queue.
2. Collect m comments from each post in the post queue, check each one of them to determine if they contain a URL and, if they do, place them in a comment queue. In all cases, the comment is added to a text file that contains all comments parsed in JSON format for later statistics.
3. For each comment in the comment queue, open the URL and search for the `iframe` tag in the raw HTML. If the page has at least one `iframe`, push it in a "to be processed later" comment queue and save the HTML along with the URL and the comment itself. If it doesn't have `iframes`, do nothing with the comment.
4. For each comment in the "to be processed later" queue, do a Google search with the comment as query followed by the word "facebook" and filter the output so that only new Facebook

pages are accepted as results.
5. Use those pages collected from the Google search in step #4 as new seed pages, and start again.

As can be seen, the Facebook graph is analyzed in a BFS fashion, in which pages and comments are first gathered, and then inspected. The number of posts and comments to be collected, called here respectively n and m, can be adjusted depending on the needs. The algorithm naturally finishes and exits when there are no more Facebook pages to crawl, but usually a termination parameter is used to shorten the search.

4.1.2 Overcoming obstacles

Before discussing the implementation details it is worth describing the problems encountered and their solutions, as they played an important part in how the structure was designed and how each step of the algorithm was implemented.

The first implementation of the crawler did not make any use of the Facebook API. Instead, the structure of the social media's pages was first carefully analyzed, and the `div` IDs and names of the containers incorporating the Facebook timeline of a page (which contains the posts along with their comments) were found. From there, posts needed to be retrieved and comments along with them. This proved to be a most difficult task, as comments are loaded for human visualization in small batches and too many clicks on the "load more comments" button were required to gather a decent amount of them. This is not only a slow process, but automating the extraction of text from the right `div` containers along with user information was taking too long in terms of implementation time at the expense of other key parts of the project. In short, the problems with this approach were many, but the most serious ones were the following:

- The `div` name containing the timeline and the subsequent `divs` holding the post IDs were not consistent between different pages.
- Only a few posts are displayed at a time, and displaying more requires pressing a button and wait for loading time.
- Clicking on a post to display comment loaded the post in a new page, causing significant memory overhead.
- Comments are loaded in very small batches for human readability purposes and are only accessible though clicks to a button. The loading is slow and inefficient.
- User names collected along with a comment cannot always be used to retrieve account information (their user ID or specific page name is needed).

The difficulty of this approach is easily explained by Facebook's need to preserve its data and make it extremely difficult for outsiders to access without their control or consent. While it is possible to write a web crawler in his way, it was beyond the scope and requirements of this project. Therefore, the Facebook API was used instead.

As already discussed in Chapter 2, Facebook does not allow free crawling of their domains and requires compliance to a set of rules and regulations, even after written permission to operate a crawler on their social media has been obtained. In order to obtain public data though the API a valid access token is required, which anyone can obtain by registering a new application within Facebook. To access private data belonging to a user or a closed group/page, the user or the admin must first accept the application and grant appropriate privileges to it. Only then the access token becomes valid for retrieving data on that particular user, group or page. As this project only required public data, a simple access token was all that was required. The access token used in this project was '1391011097865724| yZSQMjmMpFa22BlY1JIndukLaA', comprised of the application number (1391011097865724) and an actual token (yZSQMjmMpFa22BlY1JIndukLaA). The obvious downside to this is that Facebook is able to track every activity made using this access token, making the crawler absolutely non-stealthy.

Another rule relevant to this project is that they only allow 600 API calls per 600 seconds. If this is exceeded, the following error is displayed:

```
{
   "error": {
      "message": "(#613) Calls to stream have exceeded the rate of 600 calls per 600 seconds.",
      "type": "OAuthException",
      "code": 613
   }
}
```

It was therefore necessary to slow down the crawler significantly, so that this error would not be generated.

Facebook is not the only one that restricts bandwidth and server usage for performance and cost reasons. Google also blocked the crawler several times when too many queries were done. There are two different types of blocks, each of which lasts approximately half an hour:

1. Bot automated queries are blocked, but user queries made though a browser make it though if the user completes a Captcha. The user is warned that her computer is generating a large amount of automated queries and is told that it might be a virus.
2. If the above scenario is repeated many times, all queries from that IP address are blocked, and users are told they have a bot infecting their computer.

Precise measurement as of what load exactly triggers each step have not been carried though, but slowing down the crawler even further prevented these problems.

Another difficulty was dealing with JavaScript asynchronousness in PhantomJS. Being a browser, it loads pages and data in asynchronous ways which are optimized for performance, but not always easy to predict. The tasks to be completed in this project strictly needed a sequential approach. Therefore a significant amount of time was spent researching and testing for a workaround that would allow making PhantomJS execute commands in a synchronous fashion. It turns out that the solution is

recursion, as it forces the browser to call functions in a specific order rather than allowing it to decide when to schedule the next execution to achieve better performance. Most functions in the crawler had to be made recursive as a consequence of this.

4.1.3 Implementation details

In this section, details will be given as of how each part of the crawler's code relates to the algorithm description. It is worth noting that all queues mentioned below are defined as global variables, and that the crawler has been made robust by keep going when errors are encountered and continuing with the execution if a page takes too long to load (30 seconds). Moreover, the code outputs a log file though the PhantomJS console.log function, for which more details can be found in the github repository given in the Appendix.

1. **Gather n posts from each seed page and put them in a FIFO queue.**

The function **getPosts** constructs the Facebook API call to retrieve n posts from a given Facebook page, which needs to be inputted as either a page ID or by using correct name given by Facebook upon creation (which is not necessarily the same as the user name shown). The API call structure is as follows:

```
var fbgraph = 'https://graph.facebook.com/';
var access_token = '1391011097865724|9yZSQMjmMpFa22BlY1JIndukLaA';
var posts_limit = '10';

var url = fbgraph + next_page[n] + '/posts?access_token=' + access_token + '&limit=' + posts_limit;
```

In the above snippet, next_page[n] is the queue containing the seed Facebook pages. The posts limit has been fixed to 10 because it allows to gather the most recent posts while avoiding spending excessive resources on crawling one single page, allowing for greater variability and less bias of the collected data. Moreover, gathering fresher comments from the newest posts increases the chances of being able to analyze malicious pages if found, as they are more likely to still be online and not have been removed from the web. The API call returns a JSON output containing all details about the posts retrieved, such as ID, time of creation, name, message, ID and name of all users who “liked” the post, and information about all comments posted. Below a small excerpt of the output.

```
"data": [
   {
      "id": "29092950651_10154957875785652",
      "from": {
         "category": "News/media website",
         "name": "TED",
         "id": "29092950651"
      },
      "message": "\u201cYou\u2019re imperfect, and you\u2019re wired for struggle, but you are worthy of love
      belonging.\u201d",
      "picture": "https://fbexternal-a.akamaihd.net/safe_image.php?d=AQDoNnEUQfsG4E90&amp
      ;w=158&amp;h=158&amp;url=http\u00253A\u00252F\u00252Fimg.tedcdn.com\u00252Fr\u00252Fimages.ted.
      com\u00252Fimages\u00252Fted\u00252F3820be698584de25ea375c0bf57ee620caf94b8d_1600x1200.
      jpg\u00253Fc\u00253D1280\u0025252C720\u002526ll\u00253D1\u002526quality\u00253D89\u002526w\u00253D1200",
      "link": "http://t.ted.com/fnbSo2r",
      "name": "WATCH: The power of vulnerability",
```


The ID is collected from all posts and put into the post queue, ready to be used in the next step.

2. **Collect m comments from each post in the post queue, check each one of them to determine if they contain a URL and, if they do, place them in a comment queue. In all cases, the comment is added to a text file that contains all comments parsed in JSON format for later statistics.**

This is handled by the **getComments** function, which constructs the API call to retrieve comments from the pages taken from the page queue. Unlike the one for posts, the API string does not require an access token.

```
var comments_limit = '5000';
var comment_url = fbgraph + post_ids.shift() + '/comments?limit=' + comments_limit;
```

The output is again a JSON file containing in this case up to 5000 comments from each page. Increasing this number causes the crawler to slow down significantly, as more links are found and the overhead cause by each one of these is non trivial. The pages retrieved tend to be many, so enough comments are gathered in the end of each run. A sample of the output is given below.


```
"comments": {
   "data": [
      {
         "id": "10154957875785652_10154957929445652",
         "from": {
            "id": "10152923692962209",
            "name": "Laura Krogh"
         },
         "message": "This is one of my favorite ted talks.",
         "can_remove": false,
         "created_time": "2014-12-30T20:11:38+0000",
         "like_count": 63,
         "user_likes": false
      },
```


The function then saves all the comments in a text file and collects the message and searched for instances of the word 'http' or 'www.' in order to see if the message is likely to contain a URL. If the test is positive, the message itself is saved in a queue to be used in the next step.

Note how this is the first recursive function in the code. It needs to be executed for every post in the post queue, so instead of repeating the code in a loop, the following handler function is used:

```
function commentsHandler(){
    console.log("STARTING COMMENTS HANDLER");
    if(post_ids.length > 0) {
        getComments(commentsHandler);
    }
    else{
        if(post_ids.length == 0){
            console.log('number of comments to check: ' + comments_to_check.length);
            console.log('READY FOR HTML SEARCH');
            htmlHandler();
        }
    }
}
```

The function getComments(callback) calls back the commentsHandler when it is done running. The handler then checks if there are still posts for which the comments have not been extracted and analyzed. If that is the case, it calls again getComments, otherwise it starts the next step. This approach will be used for all the following functions described in this section.

3. **For each comment in the comment queue, open the URL and search for the `iframe` tag in the raw HTML. If the page has at least one `iframe`, push it in a "to be processed later" comment queue and save the HTML along with the URL and the comment itself. If it doesn't have `iframe`, do nothing with the comment.**

The function **iframeCheck** (along with its recursive handler function) performs this task. While there are still comment messages in the queue, it extracts the URL from each using a JavaScript regex. The URL is then opened and its HTML code extracted. If the HTML contains any "`<iframe`" tag opener, the HTML is considered at least vaguely suspicious and is recorded for later analysis, along with the page URL and the original comment's message. The message is then stored in a "comments to be searched" queue, which will be used in the next step of the algorithm.

Note how the check is only performed for inline `iframes` and not for all `iframes` (for example script generated). Including non inline `iframes` produces a far larger number of pages to check, many of which would only be false positives as clickjacking attacks are much more likely to be performed with static elements. Moreover, only the comment's message is recorded. This is a design choice made for simplicity purposes and to avoid cluttering in the output, with the premise that if it is absolutely necessary to have more details on which users posted such messages, a simple manual Google search can be performed to find number and quality of pages containing the same comment. Alternatively, the comments file contains a complete record of all collected comments, and it is straightforward to perform a match with the message to find out who the original poster(s) were. Finally, as suggested in the literature search, the user agent was changed to avoid pages selectively loading a different element based on browser discrimination, and was set as follows:

```
page.settings.userAgent = 'Mozilla/5.0 (X11; Linux x86_64; rv:35.0) Gecko/20100101 Firefox/35.0';
```

4. **For each comment in the "to be processed later" queue, do a Google search with the comment as query followed by the word "facebook" and filter the output so that only new Facebook pages are accepted as results.**

The **googleSearch** function and its handler take care of this step. Each comment message is searched in a Google query created as follows:

```
var url = 'http://www.google.com/search?q=' + comment + '+facebook';
```

The Google output is then filtered with a regex to find links to other Facebook pages which contain the comment somewhere in their posts. The page name is extracted and saved in the pages queue to be used in the next step of the algorithm.

5. **Use those pages collected from the Google search in step #4 as new seed pages, and start again.**

After exhausting all messages in the "comments to search" queue, the Google search handler calls back the main function **crawl**, outlined below.

```
function crawl() {
    n++;
    console.log('n = ' + n);
    if(n <= loops & n < next_page.length){
        getPosts(function(){
            console.log('posts ids found: ' + post_ids);
            commentsHandler()
            console.log('comments to check: ' + comments_to_check);
        });
    }
    else {
        console.log('THE END.');
        phantom.exit(0);
    }
}
```

The above snippet shows how writing synchronous code in JavaScript and PhantomJS can get quite tricky. The variable **loops** indicates the maximum number of pages that can be analyzed (and thus complete "loops" of the algorithm). Along with the termination criteria, the code also stops if all pages in the page queue have already been analyzed (n < next_page.length). If those conditions are not satisfied, the algorithm keeps running and the function **getPosts** is called again with another function as input. Such function will then be used as a callback when getPosts complete its execution. As can be seen, the **commentsHandler** is up next: this is the handler function responsible for calling **getComments** and collecting the comments from the pages, as explained in step #2.
If either termination criteria has been met, PhantomJS is simply closed with no error.

4.2 Data analysis

This section covers the analysis of the HTML data collected by the web crawler. The aim is to identify

clickjacking attacks by creating appropriate features and applying them to the static HTML.

As explained in chapter 2 (Context), the paper from Canali et al. [23] performs a mostly static analysis of HTML pages to detect malicious pages in general. To achieve this, they identified key features and trained several classifiers on a portion of data that was labeled, and then applied the model to the unlabeled data.

The approach here is similar. However, some key differences in this project are the focus on clickjacking (rather than generic attacks) and the origin of the data. Social network data contains page, comments and user information which can be useful for classification purposes. The feature set is therefore very different from the paper's and will be discussed in the following section. The implementation of the algorithm will be then explained afterwards.

4.2.1 Feature selection

Following a thorough analysis of clickjacking attacks and manual exploration of the crawler's output, 21 features were selected to best cover most clickjacking cases and distinguish malicious pages from benign ones. These can be categorized into four main types: **page-related** (1 to 4), **comments-related** (5 and 6), **iframes-related** (7 to 13) and **URL-related** (14 to 21).

1. **Number of times a suspicious word appears in a JavaScript inline script.** A bag of words usually correlated with malicious purposes was created, and the script is parsed to find any occurrence of such terms. The words selected are the following:

    - 'window': used to see if a script interacts with the browser window in an attempt to modify its current state.
    - 'iframe': to detect if a script either creates a new `iframe` or changes existing inline `iframes`.
    - 'mouse' and 'click': most JavaScript functions that deal with user's mouse events contain these words [27]. Tracking user movements could be not entirely malicious, but given the biased data sample (coming from random links posted as bait on Facebook pages) is a good indicator or possible problems.
    - 'track': if the script contains this word there are definitely user activity tracking components.
    - 'eval': function used to execute code. Literature review even suggested extracting the length of functions contained inside eval calls to be used as a feature for maliciousness detection [23], but it was thought to be overkill for this project.

    A bag of words approach was preferred over creating one feature for each word, as it was felt that they carry an equal weight in determining whether a script is potentially malicious or not.

2. **Number of videos in the page.** Videos and clips are excellent clickjacking baits, so it is useful to know how many are contained in the HTML, ready to be displayed to the user [22].

3. **Number of `iframes` in the page.** Having only one `iframe` and triggering other features might indicate a very high chance of clickjacking. On the other hand, if too many `iframes` are present, some features are not actually that representative of clickjacking attacks anymore, so it is important to know how many `iframes` are inserted in the page.

4. **Number of inline scripts in the page.** Imported scripts are not considered for simplicity of retrieval and analysis.

5. **Number of suspicious words in comment and URL.** As the comment is the number one bait, most attackers include a number of catchy words that is appealing to certain users. These are most often correlated with sexual content, money, weight loss programs and celebrity news, and the URL might contain them too. The terms used are the following: kilos, kg, weight, porn, sex, hot, dirty, naked, naughty, money, click, video, movie, earn.

6. **Length of comment text that accompanies the URL.** As explained above, text might help convince the user to open the included link and land on a malicious page, and longer messages might indicate the need for an attacker to further convince the user to click and/or provide explanations for the user on how to interact with the linked page.

7. **Number of `iframes` not visible to the user.** The parameters used to determine if an `iframe` is hidden from the user are: `visibility: hidden, z-index < 0` and `opacity < 0.2`.

8. **Number of times height and width are equal to 1 in an `iframe`.** Observation suggests this is quite common for hidden `iframes` to have such setup, especially those that are made to move along with the user's cursor.

9. **Number of times height and width are equal to 100% in an `iframe`.** In a classic clickjacking attack the malicious page is not visible, as the attacker wants to fool the user into thinking the `iframe` content is the actual page. Therefore, an `iframe` that covers the entire page is suspicious, although this is commonly seen in popup-like windows.

10. **Number of times an `iframe` has area smaller than 200.** The same explanation for point 8 applies.

11. **Number of `iframes` with an absolute position.** This means they can be positioned anywhere in the page, rather than within other elements such as `divs`. This allows attackers to place an `iframe` more easily where the user will click it, and to place an `iframe` with 100% width and height in a way that completely overlaps with the malicious page.

12. **Number of times an `iframe` contains a source that does not have the same domain as the page URL.** An `iframe` is much more likely to be benign if its content is strictly correlated with the page itself.

13. **Number of `iframes` that have Facebook-related content.** Examples are the Facebook like and share plugins, which are commonly found in most pages. A page without a share plugin is unlikely to spread easily within social networks, though often social engineering is used to make users post malicious content manually in other pages or in their timeline in exchange for a reward of some kind.

14. **Page URL does not contain "www.".** Most well known and established benign pages have this popular prefix.

15. **Length of URL.** Malicious pages often have either very long or very short URLs.

16. **Length of URL subdomain.**

17. **Whether the URL contains an IP address.** If yes, it is likely temporarily hosted on a private machine and more likely to be malicious.

18. **Whether the URL is shortened with a known advertisement-based shortening company.** These links first lead to the company's advertisement page, and after a certain amount of time (most often 10 seconds) the user is allowed to move on to the actual page. The page owner earns a small amount of money in the process, hence making it a very popular option among less benign pages. The most popular are **adf.ly** and **sh.st**, both of which are well known for displaying malicious content to the public [27, 28]. The paper from Nikiforakis et al. [29] discusses in details the security problems of ad-based URL shortening services.

19. **Whether the URL is shortened with a known non-ad-based shortening company.** Many malicious pages opt for shortening the URL in order to mask the true nature of their page from the user.

20. **Whether the subdomain contains numbers.** Most benign well know pages do not, but it is popular among malicious pages.

21. **Number of times the comment appears in the dataset.** Successful scams (and clickjacking) tend to spread easily in social networks. Finding the same page and comment often could mean more users have been tricked into posting it, or attackers are putting serious effort in advertising their page.

4.2.2 Implementation

As outlined before, the analysis part consisted of three different steps:

1. Extraction of the features from the data to create a feature matrix.
2. Labeling part of the data to construct a training set.

3. Training a supervised learning classifier on the labeled data, so that it can be used to separate malicious pages from benign ones and predict instances of clickjacking.

Step 1 was achieved using a Python script, which can be found in the Appendix under the name **featuregen.py**. The HTML page is retrieved along with its corresponding URL and the Facebook comment from which the crawler collected it. A counter keeps track of how many times each comment appears in the dataset.
The HTML is then parsed using the Python library BeautifulSoup, which allows to easily extract all `iframes`, scripts and videos in the page. The comment is checked to determine if it contains suspicious words and if it has text along with a URL.
The `iframes` are then analyzed separately in order to collect information about whether they might be used for malicious purposes. Height, width, area, position, opacity, z-index, source and content are checked for suspicious setups corresponding to clickjacking instances, and the relative features triggered accordingly.
Finally, the URL is analyzed to determine whether it has been shortened or not (and if the shortening was done in an ad-based fashion), its length and whether it contains an IP address or not. Its subdomain is extracted and searched for numbers and its length recorded.

After extracting the features from the data, a portion of it needed to be labeled in order to build a training set. Three labels were chosen: clickjacking (1), malicious (2) and benign (3).

An attempt at automating the process was done by constructing a static filter, which tagged each page as either 1, 2 or 3 depending on whether certain conditions were satisfied. Several versions have been written and considerable effort was spent in trying to make it more accurate, but being it somewhat naive its usage is limited to assisting a manual tagging process. The filter's conditions were the following:

- A page is benign if: it has no suspicious terms in the script, its URL is not shortened, despite having one Facebook plugin (used for liking or sharing) it only appears once in the dataset, has a subdomain length of more than 2 and the subdomain itself does not contain numbers (so that "www." can be included), has no suspicious words in the comment, the URL does not contain an IP address and has no invisible `iframes` (which is questionable, as even Amazon does).

- If the page does not satisfy the benign requirements, it is tagged as malicious.

- If the page is already tagged as malicious and: has at least one `iframe` with width = 100% and height = 100% and whose source domain is not the same domain as the URL (that is, external content), has at least one iframe with width = 1 and height = 1 and has at least one invisible iframe and a script contains suspicious terms, occurs more than 3 times in the dataset and contains some sort of Facebook plugin and contains at least one invisible iframe and there aren't too many `iframes` in the page (due to the fact that if there are, the probability that the `iframe` containing the Facebook plugin also is invisible is far lower), the URL is not shortened in an ad-based fashion (because these already known to be malicious).

With the help of the filter, 700 of the most fresh entries in the dataset were manually analyzed one by one and labeled according to the following criteria:

- Clickjacking, if the user was clearly tempted to click on something that produced unwanted and unexpected consequences.
- Malicious, if the page contained other types of scams, such as suspicious download links, request for acceptance of questionable Facebook applications, insertion and submission of private user details (for example in exchange for free phone recharges, money, hacking your friend's account etc...) or little to no content and many advertisements, often placed in positions that make them look like part of the page and are very easy to click on.
- Benign, if a page has absolutely no trace of any of the above and/or it belongs to well known, established companies such as Amazon, Google, Twitter etc...

For step 3, another Python script (**learn.py**)was used in order to take advantage of the excellent Python library scikit-learn. As will be explained in the following section, an Extra Random Trees classifier was chosen for the task and the corresponding sklearn implementation used [32]. The model was first trained on the entire labeled data and then applied to the remaining portion to obtain the results discussed in the following chapter.

4.2.3 Classifier selection

Before proceeding, a more in depth description of the data is needed. The full dataset contains 9060 entries, each of which has 21 independent features, as described in the previous section. The newest 700 entries were labeled to compose the training set, while the remaining 8360 were yet to be classified. Out of those 700 data points, 14 were clickjacking, 368 malicious and 318 benign.

The table below shows the mean and standard deviation for each feature:

	Labeled data (700)		Remaining data (8360)	
	Mean	Std	Mean	Std
1	0.18	0.3878	0.0967	0.3928
2	0.0114	0.1599	0.0188	0.1851
3	5.23	14.6829	5.0899	5.3184
4	0.22	0.4142	0.1071	0.3093
5	0.2057	0.4042	0.3052	0.4604
6	99.7857	179.5692	136.3754	1451.7964
7	0.7742	13.6455	0.2981	1.3247
8	0.0	0.0	0.0	0.0
9	0.2528	0.6635	0.0847	0.3933

10	0.6328	1.3900	0.7869	1.6285
11	0.0	0.0	0.0	0.0
12	1.7842	2.8325	1.7129	2.1583
13	1.8471	13.8781	1.4028	2.1296
14	0.7057	0.4557	0.7332	0.4422
15	47.5771	41.6142	51.9122	58.1557
16	6.0928	4.9530	6.8428	5.1019
17	0.0	0.0	0.0001	0.0107
18	0.2271	0.4189	0.0842	0.2777
19	0.0428	0.2025	0.0522	0.2225
20	0.0671	0.2502	0.1144	0.3183
21	2.0542	2.9956	1.4772	3.4163

It can be seen how several features have a high standard deviation, which indicates a noisy dataset with a significant number of outliers. With this in mind, classifiers relying on support vectors were unlikely to perform well even after scaling and normalizing the data. Nonetheless, a Support Vector Machine (SVM) was trained on the scaled and normalized training set. The labeled data was split into a single training and validation set in a random deterministic way (that is, using a seed value for the random number generator). Different split proportions were tested, and 0.25, 0.5 and 0.75 train-test split were used as indication of a classifier's performance. Although there are three labels to be chosen, we are most interested in clickjacking. Therefore k-fold cross validation was not deemed useful, as dividing the already small dataset into even smaller folds would not give the classifier enough data to detect this particular attack.

With this setup, the parameters for the SVM were tuned until the best overall accuracy was obtained. The best performing parameters were: radial basis function (RBF) or Gaussian kernel, gamma (RBF width) = 0.01, cost parameter C = 100 (which allows for more features to be selected), and class weight that reflects the distribution of the dataset (to balance the fact that clickjacking instances are very few).

The results for both splits are as follows:

	Train–test 0.25	Train–test 0.5	Train–test 0.75
Test entries	525	350	175
Validation entries	175	350	525
Total correctly classified	119	240	339
Total incorrectly classified	56	110	186
Overall accuracy	68.0	68.571	64.571

Clickjacking false positive	3	6	6
Clickjacking false negative	2	4	7
Correctly classified clickjacking	0	0	0
Total number of clickjacking	2	4	7
Number of predicted clickjacking	3	6	6

As expected, the SVM performed poorly on this dataset, as it did not fit the data well enough to detect the clickjacking instances in the validation set.

Ensemble methods such as Random Forest were therefore considered for their ability to easily absorb noisy data and their simplicity of use, as they do not require data preprocessing, perform an internal feature selection to maximize the outcome and are easy to tune. Ensemble methods combine a group of "weak learners" in order to form a "strong learner."

From the sklearn website: "A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting." [30]. The weak learners are the individual decision trees, each of which is trained with different parts of the data. Decision trees were thought to be particularly fitting to the clickjacking detection problem, as their structure resembles a "flowchart" in which the internal nodes represent a "test" on a particular attribute, the branches the outcome of the test and each leaf node is a class label, which is obtained after all attributes have been computed. The classification rules are then represented by the path from root to leaf [31]. For example, let's assume presence of a hidden `iframe` and a suspicious script are the only features, and if they are positive a page can be classified as clickjacking. A decision tree would look as follows:

Node 1: "Does this page contain hidden `iframes`?" ---- yes ----> Node 2: "Does this page have a suspicious script?" ---- yes ----> Leaf node: "Is this clickjacking?" If yes, set 1.
When fed data that corresponds to these parameters, the decision tree will output 1.

The strong learner is then formed by averaging the predictions of the individual trees. When tuning the classifier with the setup described beforehand, the best results were obtained when the number of weak estimators corresponded to roughly double the feature size, each decision tree was not limited by a maximum depth and all features were used. In particular, using extremely randomized trees (ExtraTrees classifier) instead of the classic RandomForest approach helped reduce overfitting. The table below outlines the results for the ExtraTrees classifier in the same format used for the SVM results:

	Train–test 0.25	Train–test 0.5	Train–test 0.75
Test entries	525	350	175
Validation entries	175	350	525

Total correctly classified	167	326	453
Total incorrectly classified	8	24	72
Overall accuracy	95.428	93.143	86.285
Clickjacking false positive	0	3	16
Clickjacking false negative	1	2	4
Correctly classified clickjacking	1	2	3
Total number of clickjacking	2	4	7
Number of predicted clickjacking	1	5	19

As can be seen from the 0.75 split column, the classifier was starting to overfit the data and produce a higher number of false positives. However, across all splits the number of correctly classified clickjacking entries is far better than what obtained by using SVMs. False positives rate is also more promising, being it respectively 0, 0.00859 and 0.02996 for 0.25, 0.5 and 0.75 train-test splits against 0.01704 (same for all splits) for SVMs. Moreover, the overall accuracy is much higher, which means that more malicious and benign entries have also been classified correctly. Therefore, ExtraTrees were selected as the best classifier for this task.

5. Results and evaluation

5.1 Crawler performance

The web crawler mostly performed as expected, gathering an average of 40 MB of HTML data every hour and using no more than 3 GB of RAM during execution. Its speed could not be increased any further in order not to be blocked by Google and Facebook.

A total of 1.5 GB of HTML data was collected, along with all comments encountered. However, the high number of pages opened and closed triggered a known bug in PhantomJS [33] that caused it to crash in the middle of execution, making it unlikely that any given run of the program would collect more than 150 MB of data. The crawler had to be manually restarted with new seed pages after every crash, which greatly slowed data collection and made it difficult to use on a Virtual Machine. The bug seems to have been resolved in the latest PhantomJS version, but due to dependency incompatibility in the Linux OS used to gather the data this new version could not be used.

5.2 Model results

Results of the classifier on the unlabeled data points will be covered in this section. Out of 8360 entries, 118 were labeled as clickjacking, 4465 as malicious and 4047 as benign. Recall that in the training data, out of 700 entries 14 were clickjacking, 368 malicious and 318 benign. As can be seen from the table below, the label distribution in the predictions reflects the one in the training set.

	Clickjacking	Malicious	Benign	Total
Training set	14 = 2%	368 = 52.57%	318 = 45.42%	700 = 100%
Unlabeled set	118 = 1.4%	4465 = 53.4%	4047 = 48.4%	8360 = 100%

As the focus of this project is on clickjacking attacks, the evaluation of malicious and benign pages will not be covered for a variety of reasons: partly because it is a problem already addressed in the papers covered in the Context chapter, then because in the training set the accuracy levels were extremely high (> 85%) and finally checking manually 8360 entries is an extremely daunting task, surely not fit for a project of this nature.

The 118 data points labeled as clickjacking were analyzed manually one by one to determine whether they were false positives and what kinds of clickjacking attacks were found. The false positives percentage was found to be 39.83%, while the remaining 60.16% was correctly classified.
In actual numbers, this mean that a total of 15 pages were found to be genuine, 32 malicious and 71 clickjacking. Out of the latter:

- 13 required the user to share the page on Facebook before being able to see the promised content (most often what seems to be a video or a particularly scandalous image). After sharing, they would realize said content did not exist.
- 2 attempted to make the user's browser open external software upon clicking on a benign looking page element.
- 16 used buttons usually meant for other purposed (such as the "play" or the "next" button) to redirect the user to other malicious pages.
- 8 attempted likejacking by having an `iframe` follow the user's cursor anywhere on the page. They were triggered by double clicks.
- 32 were removed from the Internet for their malicious content.

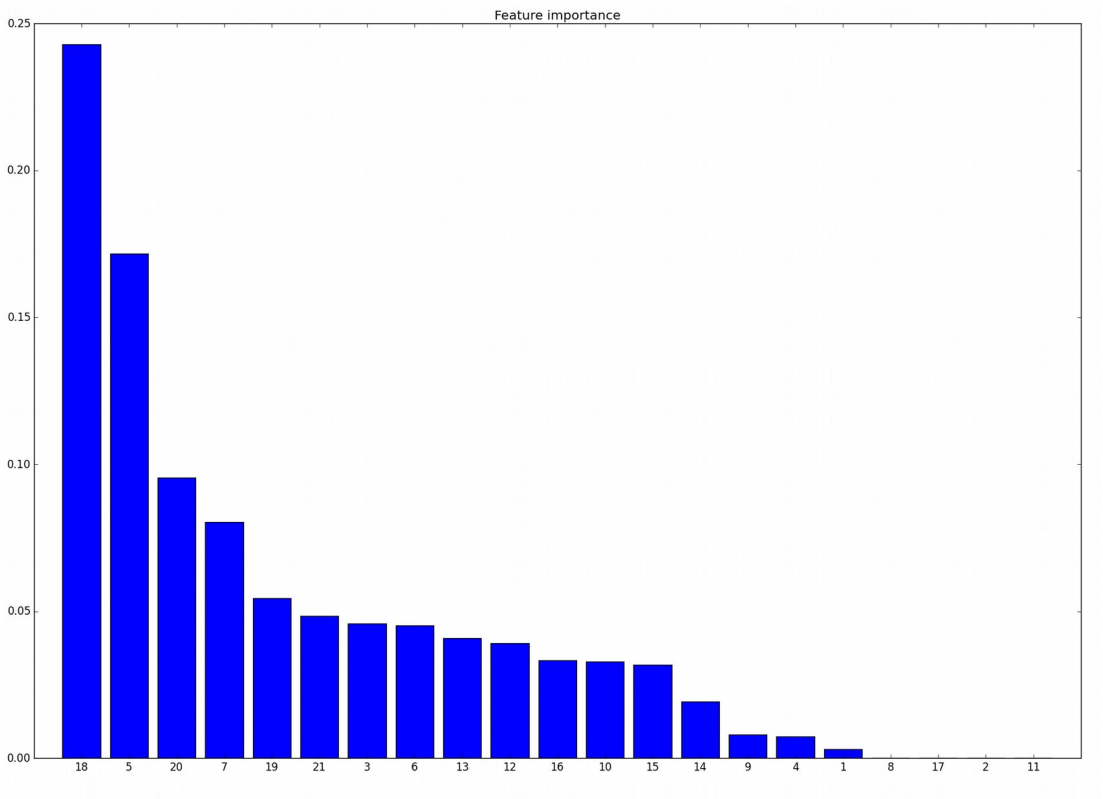
The 32 malicious and 15 benign pages were mostly incorrectly classified because they made heavy use of `iframe` for advertisements, embedding content and social media plugins. Out of the incorrectly classified malicious pages, 5 offer questionable downloads to the user, 7 are scams and the remaining ones are spam pages, that is they are mostly empty and full of advertisements.

Among the benign ones, the only major genuine page encountered is GoFundMe (6 out of 15), a website for do-it-yourself fundraising, which makes heavy use of Facebook and other social media plugins to allow its users to more easily advertise their cause to a large public. The remaining 9 pages were news websites or smaller retail services.

While a detailed analysis of clickjacking false positives was possible due to the low amount of entries to be checked, false negatives could not be evaluated manually due to the far larger amounts of data. False positives rate could not be calculated for the same reason. However, from the training set analysis, which had a 0 and 0.00859 rate respectively for 0.25 and 0.5 train-test split, it can be inferred that it would be an acceptable value. In fact, assuming all other entries have been classified

correctly as non-clickjacking, the model achieves a false positive rate of 0.0056, or 0.5%.

That said, feature analysis can be used to understand what the classifying algorithm learnt and how it relates to the results. The following is the feature ranking for the classifier trained on the full labeled data, given first in graphic format and then in a more detailed table:



Rank	Feature	Score	Description
1	18	0.242857	URL with ad-based shortening
2	5	0.171599	Number of suspicious terms in comment
3	20	0.095486	Subdomain contains numbers
4	7	0.080404	Number of `iframes` hidden
5	19	0.054532	URL shortened (not ad-based)
6	21	0.048417	Number of occurrences of comment in dataset
7	3	0.045815	Number of `iframes`
8	6	0.045181	Length of comment text (without URL)
9	13	0.040777	Number of `iframes` containing a Facebook plugin
10	12	0.039143	Number of `iframes` with source (src) not corresponding to page domain
11	16	0.033313	Subdomain length
12	10	0.032886	Number of `iframes` with small area (< 200)

13	15	0.031783	Length of URL
14	14	0.019246	Whether or not URL contains “www.” prefix
15	9	0.008077	Number of times height and width are 100% in an `iframe`
16	4	0.007402	Number of scripts in the page
17	1	0.003083	Number of suspicious words in scripts
18	8	0.000000	Number of times height and width are 1 in an `iframe`
19	17	0.000000	URL contains an IP address
20	2	0.000000	Number of videos
21	11	0.000000	Number of `iframes` with position absolute

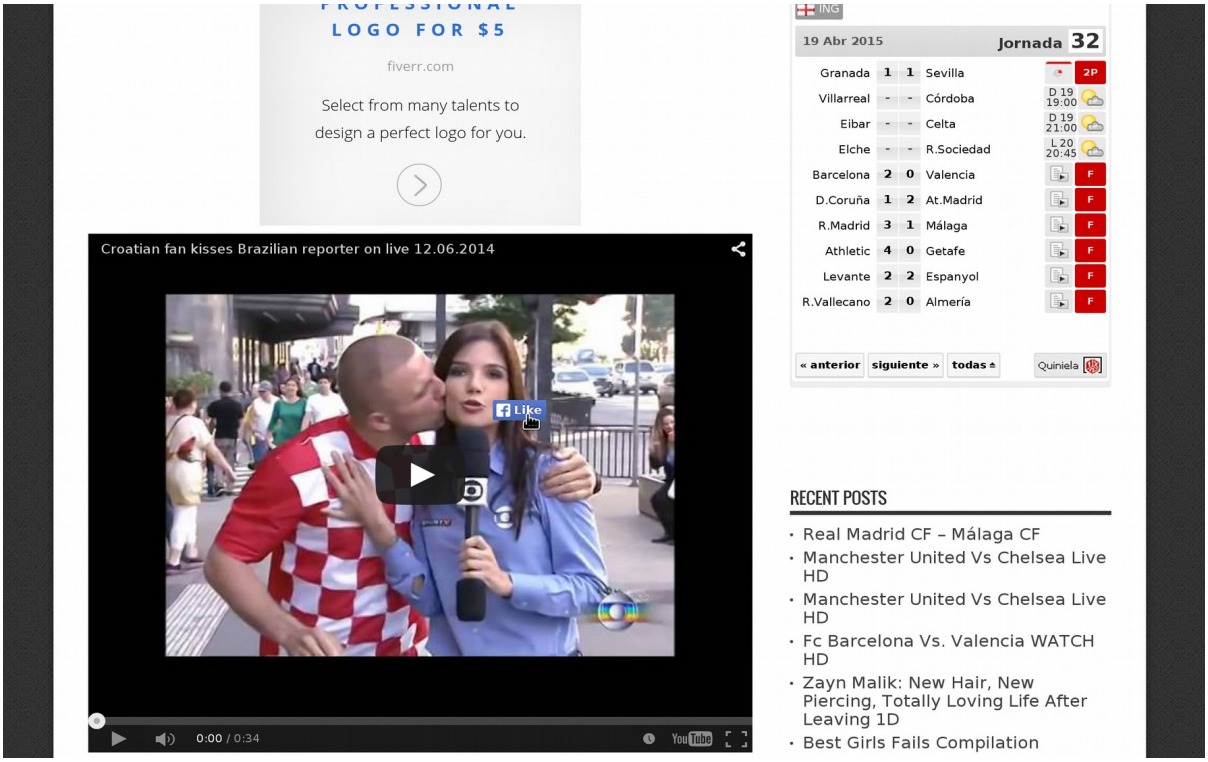
The last four features in the ranking were not found in the training set, and the classifier could not assign them an importance score. The most discriminating feature was whether the URL had been shortened in an ad-based way or not. This is because all the entries containing such features were a significant portion of the training set (159 on 700) and were all manually classified as malicious. The second best in the ranking is the number of suspicious terms in comments, which confirms how social media data can greatly help in identifying malicious websites. All other features were considered far less relevant by the classifier, though the structure of the subdomain and number of hidden `iframes` are still of significant importance.

It can therefore be safely assumed that the classifier missed several clickjacking instances, perhaps in a proportion similar to the results obtained in the validation set, if anything because the labeled data was not even enough to be representative of all the features selected. Increasing the amount of correctly classified data to be used as training set would surely be of great help in obtaining better results.

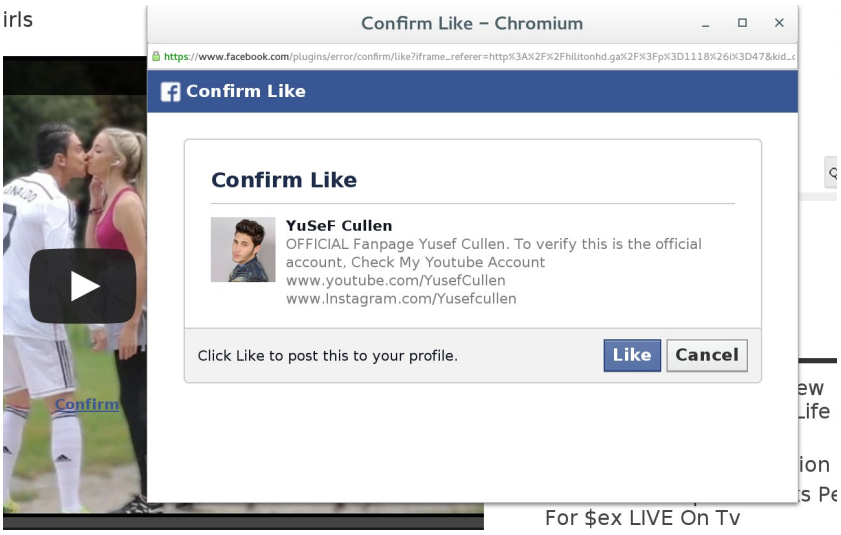
5.3 Case studies

This section will provide examples for the most common types of clickjacking detected in the dataset, ordered from the least to the most dangerous.

Likejacking



The image above shows one of the likejacking attempts. The Facebook like plugin follows the user's mouse (which turns into a hand whenever the cursor stops) and gets activated whenever a double click is performed. However, Facebook recently solved this problem by forcing the plugin to generate a popup window asking the user to confirm the like, as in the image below.



However, as they specify in the plugin's documentation [35] once the page reaches a certain level of trust the confirm window will not be displayed anymore. The data for the page displayed in Image 4 can be found below:

URL: http://tinyurl.com/15z1z5z15z1
Full comment: Pitbum ♥Cristiano Ronaldo Kissing a Fan In Public http://tinyurl.com/15z1z5z15z1
Prediction: 1 (clickjacking) with probability 0.68
Feature vector: [0, 0, 13, 0, 0, 50, 0, 0, 0, 1, 0, 6, 6, 1, 30, 7, 0, 0, 1, 0, 1]

Feature vector in Python dictionary form:
[('nsuspscript', 0), ('nvideos', 0), ('niframes', 13), ('nscripts', 0), ('ncommsuspterm', 0), ('commthastxt', 50), ('ninvisible', 0), ('nhw1', 0), ('nhw100', 0), ('nsmallarea', 1), ('nposabs', 0), ('nsrcnotdom', 6), ('nhas_fb', 6), ('nowww', 1), ('lengthurl', 30), ('subdomlen', 7), ('urlhasip', 0), ('url_adv_short', 0), ('url_short', 1), ('subdomhasnum', 0), ('noccurr', 1)]

The page contained 13 `iframes`, of which one had a very small area and 6 contained Facebook plugins of some kind. All other `iframes` had content from the same page domain. The URL is shortened in a non ad-based way and therefore contains no www. prefix.

Share to unlock content

Image 7 - Share to unlock example

Image 7 shows an example of tricking a user into sharing malicious pages on Facebook in order to unlock some kind of content. In this case, the user is presented with this scenario once she clicks on the play button. If she decides to actually share the content, she will discover that the "video" was just a bait image: the real thing does not exist. As for the other example, the data is given below:

URL: http://goo.gl/J0svAo
Comment: http://goo.gl/J0svAo
Prediction: 1 (clickjacking) with probability 0.44
Feature vector: [0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 1, 20, 3, 0, 0, 1, 0, 2]

Feature vector in Python dictionary form:
[('nsuspscript', 0), ('nvideos', 0), ('niframes', 3), ('nscripts', 0), ('ncommsuspterm', 0), ('commthastxt', 0), ('ninvisible', 0), ('nhw1', 0), ('nhw100', 0), ('nsmallarea', 0), ('nposabs', 0), ('nsrcnotdom', 2), ('nhas_fb', 2), ('nowww', 1), ('lengthurl', 20), ('subdomlen', 3), ('urlhasip', 0), ('url_adv_short', 0), ('url_short', 1), ('subdomhasnum', 0), ('noccurr', 2)]

The page contained 3 `iframes`, of which 2 had Facebook content and one was native to the page. The URL was shortened in a non ad-based way, so it does not have the www. prefix. The comment is simply the URL and it was actually found twice in the dataset.

Redirect

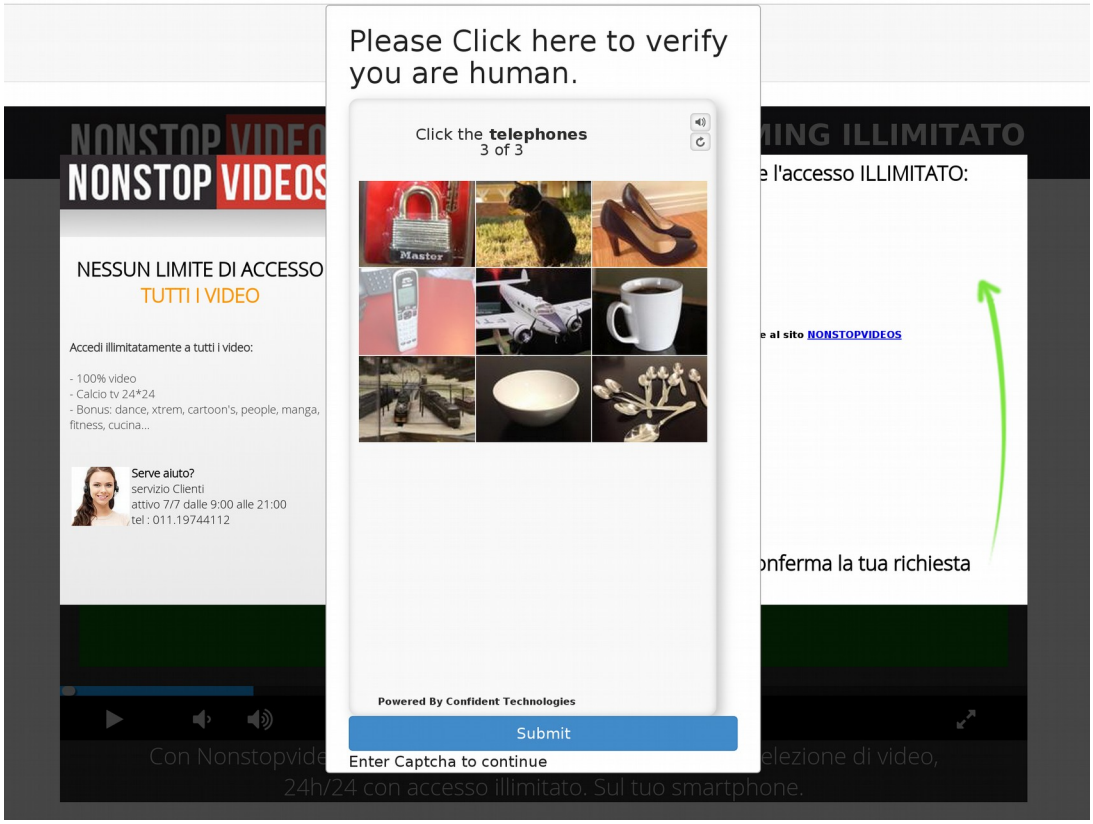


Image 5 - Redirect example #1

The user is instructed to click on certain images to "verify she is human", but every click on an image

silently opens another browser window in the background containing another malicious page.

An even more common way to perform this kind of redirects is to place the links over buttons that the user will click for other reasons. An example is the page below.

Image 6 - Redirect example #2

The page contains several image slides with various content. As soon as certain (not all) slides sets are over, this kind of page is displayed. The advertisement has been carefully placed so to look like a continuation of the current page. The user is first distracted with a "like" and "share" this page popup, then naturally tempted to click on the green arrow to proceed with the next set of slides. Doing so, however, leads to click right on the advertisement and to another malicious page. This is an extremely common setup found in many different pages. They all look the same and have mostly the same content, but different names and URLs. The data for the page in Image 6 can be found below:

URL: http://goo.gl/Pv4Fik
Comment: Shocking Celebs Then Vs. Nowhttp://goo.gl/Pv4Fik
Prediction: 1 (clickjacking) with probability 0.46
Feature vector: [0, 0, 13, 0, 0, 28, 0, 0, 0, 1, 0, 2, 2, 1, 20, 3, 0, 0, 1, 0, 1]

Feature vector in Python dictionary form:
[('nsuspscript', 0), ('nvideos', 0), ('niframes', 13), ('nscripts', 0), ('ncommsuspterm', 0), ('commthastxt', 28), ('ninvisible', 0), ('nhw1', 0), ('nhw100', 0), ('nsmallarea', 1), ('nposabs', 0), ('nsrcnotdom', 2), ('nhas_fb', 2), ('nowww', 1), ('lengthurl', 20), ('subdomlen', 3), ('urlhasip', 0), ('url_adv_short', 0), ('url_short', 1), ('subdomhasnum', 0), ('noccurr', 1)]

The container for the advertisement is a `<div>`, so the `iframes` information does not apply here. This is an excellent example, however, to show how adding a similar type of analysis for `divs` and other types of elements would be beneficial in malicious pages detection. The model classified this entry correctly because several of these examples were manually tagged as clickjacking in the training set.

6. Conclusion

This project aimed to analyze the threat of clickjacking on the Facebook social media, to understand how widespread it is and what dangers it poses to users. In order to do so, a web crawler capable of collecting data from Facebook public pages was developed, along with a method of analysis of such data in order to find clickjacking instances.

A significant amount of HTML was gathered from suspicious websites for which a link was found in the comments of the Facebook pages analyzed. Features capable of allowing a machine learning classifier to correctly identify clickjacking instances and separate malicious pages from benign ones were found and extracted from the data. A portion of the resulting feature matrix was manually analyzed and labeled to construct a training set, and an Extra Random Trees ensemble classifier trained on it. The model was then used on the unlabeled data (8360 entries), from which it predicted 118 clickjacking instances, 71 (60.16%) of which correct, 4465 malicious and 4047 benign pages.

The project's goals, namely the development of a functioning crawler and a valid data analysis method, have therefore been met. Clickjacking was found to be still an actively used attacks, and its percentage among pages shared in the Facebook social media higher to what previously found by other studies on general web pages [23], confirming the need for the development of an automated tool that would protect users from this attack.

6.1 Critics

As explained in the “Results and evaluation” chapter, both the crawler implementation and the analysis done on the data are not flawless.

The crawler is subject to a rather bad PhantomJS bug that causes it to crash too soon in the collection process, forcing the user to restart it manually every time with new seed pages. Moreover, some URLs have been found to be correlated to the wrong comments in the crawler's output, indicating the presence of another bug. Even after a thorough search, the cause of the problem could not be found and therefore could not be solved. Fortunately, however, this wrongly assigned comments are few and the problem can be fixed manually by searching for the correct comment for the link in the outputs. The crawling algorithm is good and performs well, but using Python and its excellent libraries would have surely made things far more simple and efficient, solving the need for intricate solutions in the PhantomJS code to achieve a synchronous execution. Overall, the crawler achieved what it needed to: it provided 1.7 GB of relevant HTML data to be analyzed, but there is sure room for improvement.

The analysis part had several problems. The features identified were not enough, and more are needed to identify clickjacking better. All the page elements should be checked, and a far more detailed analysis should be done on the JavaScript and the URL. For example, it is useful to determine each script's entropy to see if it has been obfuscated or not, inspect the use of JavaScript native functions, check what is being executed inside the eval() occurrences. For the URL, knowing the registration date of the page, country code, obtaining verification details for the owner and checking DNS details would be of incredibly help in detecting malicious pages.

Secondly, the training set was not big nor varied enough to allow all the features to be learned by the classifier. Manual labeling is an incredibly time-consuming process and should absolutely be avoided whenever possible. This is why writing and plugging a tool in the crawler to do an initial analysis is an absolute requirement. A large dataset of labeled clickjacking entries is needed to make this type of machine analysis effective. The Random Forest classifier chosen tends to overfit when not enough data is provided in the training process, even in its more randomized version Extra Random Trees. That, and having some comments connected to the wrong URL in the input was the main cause of the misclassifications, given that the second most important feature selected by the classifier was the number of suspicious words found in the comment.

That said, the model was still good enough to detect many clickjacking instances and correctly identify 60% of the total clickjacking predictions, which is a very promising number considering the difficulty of detection and dynamic nature of such attacks.

6.2 Future work

This project confirmed how clickjacking is still a relatively widespread problem, especially in social medias such as Facebook. It would be useful, therefore, to develop a tool capable of protecting users from such attacks. In order to do so, after solving the problems outlined in the previous section, considerable effort would need to be spent on developing an automated way to detect clickjacking during the crawling process, so to build a larger training set. With the addition of new features, the model trained in a wider and more varied dataset should be able to perform well enough to see its results integrated in a browser plugin or similar protection tool, which would then be freely distributed to the larger public.

Such a tool could be built using a combination of the detection tool plugged in the crawler, which would scan each loaded page to identify suspicious behaviors, and a database of clickjacking pages obtained from the model.

6.3 Final thoughts

This project taught how to write a web crawler to collect data and how to apply machine learning principles to analyze it, which is something of incredible value nowadays, where so many relevant things to investigate can be found on the Internet. The focus on clickjacking allowed learning more

about this relatively unfamiliar attack, and analyzing malicious links in Facebook pages led to a thorough understanding of how attackers operate in social media environments and what can be done to stop them. Therefore, it can be said that the project was indeed not only interesting, but also very useful, as many valuable skills were developed and an approach highly reusable in the real world was learned.

Bibliography

[1] Pew Research Center (2015), "Teens, Social Media & Technology", http://www.pewInternet.org/2015/04/09/teens-social-media-technology-2015/
[2] Cosenza, V (2014), "World Map of Social Networks", http://vincos.it/world-map-of-social-networks/
[3] Pew Research Center (2015), "Social Media Update 2014", http://www.pewInternet.org/2015/01/09/social-media-update-2014/
[4] Facebook (2015), "Stats", http://newsroom.fb.com/company-info/
[5] Facebook (2015), "Clickjacking", https://www.facebook.com/help/1412219392366297/
[6] OWASP (2014), "Clickjacking", https://www.owasp.org/index.php/Clickjacking
[7] Wikipedia (2015), "Clickjacking", http://en.wikipedia.org/wiki/Clickjacking
[8] OWASP (2015), "Clickjacking Defense Cheat Sheet", https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet
[9] Rydstedt, Bursztein, Boneh, Jackson (2010), "Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites", http://w2spconf.com/2010/papers/p27.pdf
[10] Zalewski, M. (2011), "Browser Security Handbook, part 2", https://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_%28UI_redressing%29
[11] Zeltser, L. (2015), "How Clickjacking Attacks Work", https://zeltser.com/clickjacking-the-next-generation/
[12] Trend Micro (2012), "Think Before You Click: Truth Behind Clickjacking on Facebook", http://about-threats.trendmicro.com/fr/webattack/108/Think+Before+You+Click+Truth+Behind+Clickjacking+on+Facebook
[13] OWASP (2015), "Cross-Site Request Forgery (CSRF) ", https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29
[14] Qualys, Inc. Security Labs (2012), "Clickjacking: An Overlooked Web Security Hole ", https://community.qualys.com/blogs/securitylabs/2012/11/29/clickjacking-an-overlooked-web-security-hole
[15] Huang, Moshchuk, Wang, Schechter, Jackson (2012), "Clickjacking: Attacks and Defenses", https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final39.pdf
[16] W3schools (2015), "HTML <iframe> Tag ", http://www.w3schools.com/tags/tag_iframe.asp
[17] Kotowicz, K. (2015), "Is this a good example of cursorjacking?", http://koto.github.io/blog-kotowicz-net-examples/cursorjacking/
[18] Wikipedia (2015), "Web Crawler", http://en.wikipedia.org/wiki/Web_crawler
[19] Catanese, De Meo, Ferrara, Fiumara, Provetti (2011), "Crawling Facebook for Social Network Analysis Purposes", http://arxiv.org/pdf/1105.6307.pdf
[20] Facebook (2015), robots.txt, https://www.facebook.com/robots.txt
[21] Canali, Cova, Vigna, Kruegel (2011), "Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages", https://www.cs.ucsb.edu/~vigna/publications/2011_canali_cova_kruegel_vigna_Prophiler.pdf
[22] Cova, Kruegel, Vigna (2010), "Detection and Analysis of Drive-by-Download Attacks

and Malicious JavaScript Code",
"http://cs.ucsb.edu/~vigna/publications/2010_cova_kruegel_vigna_Wepawet.pdf
[23] Balduzzi, Egele, Kirda, Balzarotti, Kruegel (2010), "A Solution for the Automated Detection of Clickjacking Attacks
", https://iseclab.org/papers/asiaccs122-balduzzi.pdf
[24] SeleniumHQ (2015), http://www.seleniumhq.org/
[25] PhantomJS (2015), http://phantomjs.org/
[26] Koch, P. (2015), "Mouse Events", http://www.quirksmode.org/js/events_mouse.html
[27] Google (2015), "Safe Browsing Diagnostic page for adf.ly",
http://www.google.com/safebrowsing/diagnostic?site=adf.ly/
[28] Google (2015), "Safe Browsing Diagnostic page for sh.st",
https://safebrowsing.google.com/safebrowsing/diagnostic?site=sh.st/
[29] Nikiforakis, Maggi, Stringhini, Rafique, Joosen, Kruegel, Piessens, Vigna, Zanero (2014), "Stranger Danger: Exploring the Ecosystem of Ad-based URL Shortening Services",
https://lirias.kuleuven.be/bitstream/123456789/440951/1/strangerdanger_www2014.pdf
[30] scikit-learn (2015), "RandomForestClassifier", http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
[31] Wikipedia (2015), http://en.wikipedia.org/wiki/Decision_tree
[32] scikit-learn (2015), "ExtraTreesClassifier", http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html#sklearn.ensemble.ExtraTreesClassifier
[33] GitHub ghostdriver (2014), "PhantomJS crashes after multiple page load / async script calls", https://github.com/detro/ghostdriver/issues/328
[34] Facebook (2015), "Social Plugins FAQs",
https://developers.facebook.com/docs/plugins/faqs#like-confirm

Appendix

The full code, along with a detailed description of how to run it and sample inputs and outputs can be found in the following github repository: https://github.com/sjowastaken/Final-Year-Project

---

Project plan

Name: Giulia Deiana
Supervisor's name: Gianluca Stringhini
Project title: "Analysis and detection of clickjacking on Facebook"

Aim

This project aims to collect public data from the Facebook social network and use it to train a machine learning model with the aim of detecting instances of clickjacking.

Objectives

1. Learn how to collect raw real world data by writing a successful web crawler that will comply with Facebook's rules and thus won't be detected and/or banned.

2. Analyze the problem of clickjacking and find ways to automatically detect it.

3. Create a training set and a data set from the raw data to use for a supervised learning algorithm.

4. Develop a machine learning algorithm that after being trained on the training data will be able to successfully detect clickjacking given the raw data.

Deliverables

1. A web crawler written in Phantom JS and Selenium that is able to stealthily crawl Facebook to collect public user data, saving it in text files.

2. A machine learning algorithm capable of identifiying clickjacking instances by parsing text files (dumps of web pages).

Schedule

October (4 weeks): literature review.

Novemeber – December (8 weeks) : study clickjacking, develop webcrawler and collect data.
January – March (12 weeks): develop and implement algorithm.
April – May (8 weeks): write project paper.

Project interim report

Name: Giulia Deiana
Supervisor's name: Gianluca Stringhini
Project title: "Analysis and detection of clickjacking on Facebook"

1. Progress made to date

The goal of the project is to retrieve user data from public Facebook pages in order to analyze the common problem of UI – redressing, of which a popular subset is better known as clickjacking. This project aims to understand how widespread this problem is on Facebook and how a person's profile predicts the likelihood of him/her falling for such scams.

1.1 – Obtaining permissions

Facebook does not allow crawlers and its structure and code are created in such a way that it is hard to collect data from it without being strictly monitored, even if all the data collected is strictly public. Therefore we registered the project with Facebook itself and with the UCL Data Protection Office, while also asking UCL for ethical approval. The code runs on a UCL VM that Facebook knows to be associated with this project, therefore making the collection of data absolutely legal.

1.2 – Understanding the UI – Redressing problem (clickjacking included)

The first step that needed to be done was to analyze the problem of clickjacking. This, in turn, led to its parent problem of general UI-Redressing. As the name itself suggests, the attack consists in "redressing" the UI so that the user is deceived into thinking he/she is interacting with a certain webpage when in reality there's a hidden layer underneath that reacts to user inputs to carry on malicious activities. Examples of these are posting automatically on the user's Facebook timeline (and/or spamming their friend's timelines as well or sending them spam private messages), put a "like" in certain pages on their behalf, steal their cookies, send automated emails, download viruses and infect their computers etc...

In its most common format, an attacker sets up a web page A with a hidden `iframe` in which a malicious page B is loaded. When the user clicks on some button seemingly in A (for example, the "start video" button if the bait is a video), it is page B who executes its own code without the user noticing. Sometimes the cursor of the user is followed everywhere in the page, so that whenever the

user clicks the malicious code is executed.

Common example in nowadays Facebook include scams about celebrity pictures, lies about sudden deaths or mishappenings of famous people or (mostly false) catastrophic news pictures and videos. Iframe busters are the most common solution (adding code in web pages that does not allow them to be embedded in `iframes` or that warns the user when an `iframe` is overlaid on them). It is, however, hard to detect when `iframes` are used legitimately and when they're being abused for malicious activities, so banning them entirely is not a solution.

1.3 – Selecting the right tools

In order to collect the data from Facebook, a web crawler needed to be implemented. Several tools were considered for this purpose, among which python, Selenium + PhantomJS and Google chrome. After trying out selenium (a tool that allows to automate browser activities) along with PhantomJS (a fast headless browser), selenium was dropped in favor of a browser only solution. The code for the crawler is written entirely in JavaScript and so far no extensions have been used.

1.4 – Gathering data from Facebook

Understanding Facebook structure was a crucial step in extracting data from it. After raw HTML observation and Facebook graph documentation, it was understood that in order to get user comments, the respective post's ID and the corresponding page ID had to be retrieved first.
The first attempt at extracting post IDs from Facebook was done by loading the website's JavaScript and reading raw HTML data from the downloaded content. The main difficulties with this approach turned out to be the following:

- Not all Facebook pages present the same layout, which made it very difficult to decide which DOM structure to extract the data from. Some pages contained the post IDs in the "mainContainer" `div`, others in the "contentArea", others still in the "timelineUnitContainer".

- The post IDs are scattered around and appear multiple times (for example comments do contain the post ID of the parent post), but it seemed that they all appeared under the "fbid="..." format, so the following regex was indeed successful in retrieving all of them from a given output: match("fbid=(.*)&amp").

- Comments are very hard to fetch with this method and no solution was found on how to access them using PhantomJS alone.

- Slow. Every page needs to be loaded, queried and the output scanned with a regex.

After more research, it turned out that the preferred method to analyze Facebook data is to use the '

Facebook graph', which allows developers to easily access any kind of data in the social network. The only downside is that an application ID and a token are necessary to access such service, and admittedly creating a “stealthy” crawler that leaves obvious footprints did not sound like the best idea. Public data can be viewed with any kind of app ID and token, whether privileged or not. After creating a dummy application and the correspondent identification details it was possible to note that data retrieval is much faster and cleaner with this method. The Facebook graph returns the content of the page in JSON format, which makes it easy to read and handle. The post IDs along with the corresponding post's comments are easy to collect, along with the information of the user who posted each comment and “liked” the post.

1.5 – The crawling algorithm

The crawler is first given a list of popular web pages that have a high probability of containing posts with spam comments. Then the pages are analyzed one by one, and if spam comments with a link are found, the comment is collected along with the profile information of the user who posted it. The crawler then saves the page and once all of the starting pages have been analyzed, it performs a Google search with the spam comment to see if other Facebook pages also have posts containing that comment and then starts crawling those pages, repeating the process.

2. Remaining work to be done

In the project plan, the following objectives were identified:

1. Learn how to collect raw real world data by writing a successful web crawler that will comply with Facebook's rules and thus won't be detected and/or banned.

2. Analyze the problem of clickjacking and find ways to automatically detect it.

3. Create a training set and a data set from the raw data to use for a supervised learning algorithm.

4. Develop a machine learning algorithm that after being trained on the training data will be able to successfully detect clickjacking given the raw data.

So far, point 1 and 2 have been addressed. Points 3 and 4 will be completed in the upcoming months.

Code listing

crawler.js

```
/* PhantomJS Facebook crawler */
```

```
var fbgraph = 'https://graph.facebook.com/';
var access_token = '1391011097865724|9yZSQMjmMpFa22BlY1JIndukLaA';

/* Customizable variables */
var posts_limit = '10';
var comments_limit = '5000';
var loops = 100000000;
/* Set to 0 to deactivate logs */
var verbose = 1

/* Initialize filesystem for comment and HTML writing. */
var fs = require('fs');
var commentsPath = "./data/all_comments.txt";
if(fs.exists(commentsPath)){
    fs.remove(commentsPath);
}

var htmlPath = "./data/all_html.txt";
if(fs.exists(htmlPath)){
    fs.remove(htmlPath);
}

var i = 0;
var j = 0;
var n = -1;
var urlRegex = /(https?:\/\/[^\s]+)/g;

/* Contains name of pages to be searched next_page. */
var next_page = [];
var post_ids = [];
var comments_to_search = [];
var comments_to_check = [];
var links = [];

var nextindex = 0;
var postsindex = 0;
var page_id_index = 0;

/* All seed pages used, taken from: http://fanpagelist.com/category/top_users/

 f4ep, facebook, shakira, Cristiano, eminem, VinDiesel, cocacola, michaeljackson,
 TheSimpsons, rihanna, JustinBieber, LeoMessi, BobMarley, harrypottermovie,
 WillSmith, TaylorSwift, beyonce, TexasHoldEm, linkinPark, jackie,
 manchesterunited, ladygaga, MrBean, adele, pitbull, brunomars, Selena,
 McDonaldsUS, TitanicMovie, DavidGuetta, spongebob, avrillavigne, AKON, neymarjr,
 FamilyGuy, LilWayne, MeganFox, AvatarMovie.UK, Beckham, JasonStatham, Enrique,
 usher.
*/

/* Initialize seed pages */
next_page[0] = 'f4ep';
next_page[1] = 'facebook';
next_page[2] = 'shakira';

crawl();

function crawl() {
    n++;
    if(verbose == 1) console.log('n = ' + n);
    if(n <= loops & n < next_page.length){
        getPosts(function(){
```

```
            if(verbose == 1) console.log('posts ids found: ' + post_ids);
            commentsHandler()
            if(verbose == 1) console.log('comments to check: '+comments_to_check);
        });
    }
    else {
        if(verbose == 1) console.log('THE END.');
        phantom.exit(0);
    }
}

/* Gets post IDs from a Facebook page */
function getPosts(callback) {
    var page = require('webpage').create();
    if(verbose == 1) console.log('enter crawl');
    var url = fbgraph + next_page[n].toString() + '/posts?access_token=' +    access_token
+ '&limit=' + posts_limit;

    /* Wait 30 seconds before deciding that the page is unreachable and move on */
    page.settings.resourceTimeout = 30000;

    page.onResourceTimeout = function(e) {
        if(verbose == 1) console.log('\n\nPAGE DIDN\'T OPEN\n\n');
        if(verbose == 1) console.log(e.errorCode);
        if(verbose == 1) console.log(e.errorString);
        if(verbose == 1) console.log(e.url);
        page.release();
        htmlHandler();
    }

    if(verbose == 1) console.log('opening url: ' + url);

    page.open(url);
    page.onLoadFinished = function(status) {
        if (status === 'success') {
            var data = JSON.parse(page.plainText);

            if(data.error) {
                if(verbose == 1) console.log('----- page had error, skipping it
             -----');
                if(verbose == 1) console.log(page.plainText);
                if(verbose == 1) console.log('----------------------------------------');

                crawl();
                return;
            }
            if(!data.data) {
                if(verbose == 1) console.log('no posts found in page');
                if(verbose == 1) console.log(page.plainText);
                page.close();
                crawl();
                return;
            }
            if(verbose == 1) console.log('data length = ' + data.data.length);
            for(i = 0; i < data.data.length; i++){
                if(verbose == 1) console.log('pushing data: ' + data.data[i].id);
                post_ids.push(data.data[i].id);
                postsindex++;
            }
        }
        else {
```

```
            if(verbose == 1) console.log('error crawling page ' );
            if(verbose == 1) console.log(
                "Error opening url \"" + page.reason_url
                + "\": " + page.reason
                );
        }
        page.release();

        if(verbose == 1) console.log('closing crawl page');
        page.close();

        callback();
    };

    page.onResourceError = function(resourceError) {
        page.reason = resourceError.errorString;
        page.reason_url = resourceError.url;
    };

}

function commentsHandler(){
    if(verbose == 1) console.log("STARTING COMMENTS HANDLER");
    if(post_ids.length > 0) {
        getComments(commentsHandler);
    }
    else{
        if(post_ids.length == 0){
            if(verbose == 1) console.log('number of comments to check: ' +
                                         comments_to_check.length);
            if(verbose == 1) console.log('READY FOR HTML SEARCH');
            htmlHandler();
        }
    }
}

/* Get comments from Facebook posts using post ID and save them to file. If a
   comment contains a URL, save it in a queue for it to be searched on Google
   later */

function getComments(callback) {
    if(verbose == 1) console.log('entering getComments');

    var page = require('webpage').create();
    comment_url = fbgraph + post_ids.shift() + '/comments?limit='+comments_limit;
    jsonComments = '';
    if(verbose == 1) console.log('opening comments url: ' + comment_url);

    page.settings.userAgent = 'Mozilla/5.0 (X11; Linux x86_64; rv:35.0)
                           Gecko/20100101 Firefox/35.0';

    page.open(comment_url);
    page.onLoadFinished = function(status) {
        if (status === 'success') {
            if(verbose == 1) console.log('saving comments...');
            text = page.plainText;
            fs.write(commentsPath, text, 'a');
            jsonComments = JSON.parse(text);
            if(verbose == 1) console.log('length = ' + jsonComments.data.length);

            if(jsonComments.data.length == 0) {
```

```
                if(verbose == 1) console.log('no comments found in post');
                page.close();
                commentsHandler();
                return;
            }
            for(k = 0; k < jsonComments.data.length; k++) {
                /* If message contains a url, search the whole message on google
              to see if it's been posted in other pages */
                message = jsonComments.data[k].message;
                if(message.indexOf("http") > -1){
                    comments_to_check.push(message);
                }
            }
        }
        else {
            if(verbose == 1) console.log('error in getComments page ' );
            if(verbose == 1) console.log(
                "Error opening url \"" + page.reason_url
                + "\": " + page.reason
                );
        }
        page.release();
        if(verbose == 1) console.log('closing getComments page');
        page.close();
        callback.apply();

    };
}


function htmlHandler() {
    if(verbose == 1) console.log('STARTING HTML HANDLER');
    if(verbose == 1) console.log('comments_to_check length is ' +
       comments_to_check.length);
    if(comments_to_check.length > 0) {
        iframeCheck();
    }
    else {
        if(comments_to_check.length == 0){
            searchHandler();
        }
    }
}

/* Check if a given webpage contains inline iframes */
function iframeCheck() {
    if(verbose == 1) console.log('STARTING IFRAMECHECK');
    comment = comments_to_check.shift();
    if(verbose == 1) console.log('\nchecking comment: ' + comment);

    var page = require('webpage').create();
    page.settings.resourceTimeout = 30000;

    page.onResourceTimeout = function(e) {
        if(verbose == 1) console.log('\n\nPAGE DIDN\'T OPEN\n\n');
        if(verbose == 1) console.log(e.errorCode);
        if(verbose == 1) console.log(e.errorString);
        if(verbose == 1) console.log(e.url);
        page.release();
        htmlHandler();
    }
```

```
    page.onResourceError = function(resourceError) {
        page.reason = resourceError.errorString;
        page.reason_url = resourceError.url;
    };

    var url = comment.match(urlRegex);
    if(url != null) {
        url = url.toString();

        if(url.match(/www\.facebook/) === null & url.match(/www\.youtube/) ===
       null & url.match(/m\.facebook/) === null & url.match(/m\.youtube/) ===        null &
url.match(/play\.google/) === null & url.match(/youtu\.be/) ===         null) {
            if(verbose == 1) console.log('this link passed the test: ' + url);
            if(verbose == 1) console.log('opening url: ' + url);

            page.settings.userAgent = 'Mozilla/5.0 (X11; Linux x86_64; rv:35.0)
                                       Gecko/20100101 Firefox/35.0';

            page.open(url, function(status) {
                if(verbose == 1) console.log('iframecheck page loaded');
                if (status === 'success') {
                    var html = page.content;
                    if(html != null){
                        if(html.indexOf("<iframe") > -1) {
                            if(verbose == 1) console.log('writing html for page '
                                                    + url + '...');
                            fs.write(htmlPath, '\n\n\n-------- HTML from page: ' +
             url + '.\n\ncomment is: ' + comment + '\n\n' + html, 'a');
                            if(verbose == 1) console.log('done writing html');
                            comments_to_search.push(comment);
                            if(verbose == 1) console.log('pushed comment: ' +
                                                    comment);
                        }
                    }
                }
                else {
                    if(verbose == 1) console.log('error in iframecheck page ' );
                    if(verbose == 1) console.log(
                        "Error opening url \"" + page.reason_url
                        + "\": " + page.reason
                        );
                }
                page.release();
                if(verbose == 1) console.log('closing iframecheck page');
                page.close();
                htmlHandler();
            });
        }
        else {
            if(verbose == 1) console.log('url didn\'t pass the test');
            page.release();
            htmlHandler();
        }
    }
    else {
        if(verbose == 1) console.log('url null');
        page.release();
        htmlHandler();
    }
```

```
}


function searchHandler() {
    if(verbose == 1) console.log('STARTING SEARCH HANDLER');
    if(verbose == 1) console.log('comments to search length: ' +
                        comments_to_search.length);
    if(comments_to_search.length > 0) {
        googleSearch();
    }
    else {
        if(comments_to_search.length == 0){
            if(verbose == 1) console.log('next page: ' + next_page);
            if(verbose == 1) console.log('CRAWL AGAIN');
            crawl();
        }
    }
}

/* Search suspicious comments on Google and save other Facebook pages that contain it */
function googleSearch() {
    if(verbose == 1) console.log('entering googleSearch');
    comment = comments_to_search.shift();

    var url = 'http://www.google.com/search?q=' + comment + '+facebook';

    if(verbose == 1) console.log('google search for comment: ' + comment);

    var temp = '';
    var page = require('webpage').create();

    page.settings.userAgent = 'Mozilla/5.0 (X11; Linux x86_64; rv:35.0)
                            Gecko/20100101 Firefox/35.0';

    page.open(url, function(status) {
        if (status === 'success') {
            var links = page.plainText.match(/https:\/\/www.facebook.com\/
                    (.*?)\n/g);
            if(links != null){
                for(i = 0; i < links.length; i++) {
                    temp = links[i].split("/")[3];

                    /* Remove duplicates and any entry that contains a dot.*/
                    if(temp.indexOf("pages") == -1 && temp.indexOf(".") == -1 &&
                    next_page.indexOf(temp) == -1) {
                        next_page.push(temp.replace(/(\r\n|\n|\r)/gm,''));
                        nextindex++;

                    }
                }
            }
        }
        else {
            if(verbose == 1) console.log('error in googleSearch page ' );
            if(verbose == 1) console.log(
                "Error opening url \"" + page.reason_url
                + "\": " + page.reason
                );
        }
        page.release();
        if(verbose == 1) console.log('closing googleSearch page');
```

```
        page.close();
        searchHandler();

    });
}
```

featurelearn.py

```
import re
import os
import sys
from bs4 import BeautifulSoup
import urllib.request
from urllib.parse import urlsplit
import time
import numpy as np
import cssutils
import collections

start = time.time()

#Set to 0 to deactivate logs
verbose = 1

# Saved file name
outn = 'example_output'

# Data file name
dn = 'example_all_html'

def init_fv():

       fv = collections.OrderedDict()

       fv['nsuspscript'] = 0
       fv['nvideos'] = 0
       fv['niframes'] = 0
       fv['nscripts'] = 0

       fv['ncommsuspterm'] = 0
       fv['commthastxt'] = 0

       fv['ninvisible'] = 0
       fv['nhw1'] = 0
       fv['nhw100'] = 0
       fv['nsmallarea'] = 0
       fv['nposabs'] = 0
       fv['nsrcnotdom'] = 0
       fv['nhas_fb'] = 0

       fv['nowww'] = 0
       fv['lengthurl'] = 0
       fv['subdomlen'] = 0
       fv['urlhasip'] = 0
       fv['url_adv_short'] = 0
       fv['url_short'] = 0
       fv['subdomhasnum'] = 0

       fv['noccurr'] = 0
```

```
	fv['filterpred'] = 0

	return fv

def is_url_shortened(url):
	# Check if url was shortened
	shorteners = ['goo.gl', 'tinyurl.com', 'tiny.cc']
	if any(x in url for x in shorteners):
		return 1
	else: return 0

def is_url_adv_shortened(url):
	# Check if url was shortened in ad-based way
	shorteners = ['bit.ly', 'ow.ly', 'adf.ly', 'sh.st', 'fur.ly', 'cutt.us',
		 'cur.lv', 'past.is', 'po.st', 'adyou.me', 'jota.pm', 'j.gs',
	  'q.gs']
	if any(x in url for x in shorteners):
		return 1
	else: return 0

def has_keywords(comment):
	# Find terms that more often correlate with malicious intentions
	keywords = ['kilos', 'kg', 'weight', 'porn', 'sex', 'sexo', 'hot', 'dirty',
		'naked', 'naughty', 'money', 'click', 'video', 'movie', '$ex', 'earn']
	if any(x in comment.lower() for x in keywords):
		return 1
	else: return 0

def has_susp_script(scripts):
	# Find terms that more often correlate with malicious activity
	count = 0
	for s in scripts:
		suspterms = ['window', 'iframe', 'mouse', 'click', 'track', 'eval']
		for susp in suspterms:
			if susp in s.lower():
				count += 1

	return count

def has_text(comment, comment_url):
	# Determine whether comment contains text along with url
	comment_text = comment.replace(comment_url, "").strip()
	return len(comment_text)

def inline_transp_check(ifr):
	# Check for transparency
	is_invisible = 0
	if 'visibility' in ifr.attrs:
		if ifr['visibility'] != 'visible':
			is_invisible = 1
	if 'opacity' in ifr.attrs:
		if ifr['opacity'] == '0':
			is_invisible = 1
	if 'z-index' in ifr.attrs:
		if int(ifr['z-index']) < 0:
			is_invisible = 1

	return is_invisible

def analyze_iframes(iframes, page_url, fv):
	for ifr in iframes:
```

```
w = ''
h = ''
is_invisible = inline_transp_check(ifr)

if 'style' in ifr.attrs:
	css = ifr['style']
	s = cssutils.parseStyle(css)
	w = s.width
	h = s.height
	if 'absolute' in s.position:
		cssposabs = 1
	if s.visibility == 'hidden':
		is_invisible = 1
	if s.opacity == '0':
		is_invisible = 1

fv['ninvisible'] += is_invisible

if 'width' in ifr.attrs:
	w = ifr['width'].strip(' ')
if 'height' in ifr.attrs:
	h = ifr['height'].strip(' ')

w = re.findall(r'\d+', w)
h = re.findall(r'\d+', h)

if len(w) > 0 and len(h) > 0:
	w = int(w[0])
	h = int(h[0])
	area = -1
	if(w == 0 and h == 0) or (w != 0 and h != 0):
		area = w * h
		if area < 250:
			fv['nsmallarea'] += 1
		elif area == 1:
			fv['nhw1'] += 1
		elif area == 10000:
			fv['nhw100'] += 1

if 'src' in ifr.attrs:
	src = ifr['src']
	if "facebook" in src:
		fv['nhas_fb'] += 1

	# Benign domains
	benigndom = ['accounts.google.com', 'youtube.com', 'ulogin',
		'blogger.com', 'wix', 'addtoany.com']
	if not any(x in src for x in benigndom):
		# Same domain check
		base_src = urlsplit(src).netloc
		base_url = urlsplit(page_url).netloc
		if base_src != base_url:
			if 'id' in ifr.attrs:
				# Don't pick up google iframes, developers try
				  to hide them
				if 'google' not in ifr.attrs['id']:
					fv['nsrcnotdom'] += 1
if 'position' in ifr.attrs:
	if ('absolute' in ifr['position']) or cssposabs == 1:
		fv['nposabs'] += 1
```

```
	return fv

def analyze_url(url, fv):
	# See if subdomain is www
	spl = (url.split('://')[1]).split('.')
	subdom = spl[0]
	fv['lengthurl'] = len(url)
	# Whitelisting twitter
	fv['subdomlen'] = len(subdom)
	if subdom != 'www' and subdom != 'twitter' and spl[1] != 'tumblr' and subdom !=
'youtu':
		fv['nowww'] = 1
	# Find if it contains ip address
	ip = re.findall( r'[0-9]+(?:\.[0-9]+){3}', url)
	if len(ip) > 0:
		fv['urlhasip'] = 1
	fv['url_adv_short'] = is_url_adv_shortened(url)
	fv['url_short'] = is_url_shortened(url)
	if any(char.isdigit() for char in subdom):
		fv['subdomhasnum'] = 1
	return fv

def labelgen(fv):

	'''
	1: Clickjacking: is malicious AND:
	- one iframe only and w == 100 and h == 100 and srcnotdom
	- w == 1 and h == 1 and is_invisible and script suspicious
	- noccurr > 3 and is_fb true and is_invisible true and there aren't too many
	iframes to skew up the result
	- urladshort is not true

	3: Benign pages
	- no bad terms in script
	- url not shortened in any way (urladshort and urlshort == 0)
	- (is_fb == 1 and appears only once in dataset (noccur == 1))
	- subdomain present (subdomainlen > 2) and subdomain has no numbers
	- no bad terms in comment (nocommsuspterm == 0)
	- no ip address in URL (hasipaddr == 0)
	- is_invisible never triggered

	Else: 2, malicious

	'''

	label = 0

	if fv['nsuspscript'] <= 1 and (not (fv['noccurr'] > 2 and fv['nhas_fb'] >
	0)) and fv['url_adv_short'] == 0 and fv['url_short'] == 0 and
		fv['ncommsuspterm'] == 0 and fv['urlhasip'] == 0 and (fv['subdomlen'] >
	      2 and fv['subdomhasnum'] == 0) and fv['ninvisible'] == 0:
		label = 3
	else:
		label = 2
		if fv['url_adv_short'] == 0 and ((fv['nhw1'] > 0 and fv['ninvisible'] >
		     0 and fv['nsuspscript'] > 0) or \
		       (fv['noccurr'] > 3 and fv['nhas_fb'] > 0 and fv['ninvisible'] > 1
		and fv['niframes'] <= 20) or \
		       (fv['nhw100'] == 1 and fv['nsrcnotdom'] > 0)):
		       label = 1
```

```
	return label

def process_pages(pages):
	log = {}

	htmls = []
	page_urls = []
	comments = []
	log_entries = []

	for i in range(0, len(pages)):
		# Early termination
		#if i > 100: break
		if i % 100 == 0: print("processed ", i, " pages")
		html = pages[i].split("<head>")

		if(len(html) > 1):
			nonhtml = html[0].split("comment is: ")
			page_url = nonhtml[0].split("from page: ")[1].split(" ")
				[0].strip()
			page_url = page_url[:len(page_url)-1]

			# Get comment
			comment = nonhtml[1].split("<!DOCTYPE")[0]
			log_entry = page_url + " " + comment.replace('\n', '')

			if log_entry not in log:
				log[log_entry] = 1
			else:
				log[log_entry] += 1

			htmls.append(html)
			page_urls.append(page_url)
			comments.append(comment)
			log_entries.append(log_entry)

	return htmls, page_urls, comments, log_entries, log


# --- SCRIPT

pages = open(''.join((."./data/", dn, ."txt"))).read()
if verbose == 1: print("opening file: ", ''.join((."./data", dn)))
pages = pages.split("-------- HTML ")

featurevectors = []
already_processed = []

# Counts non duplicate pages
count = 1

htmls, page_urls, comments, log_entries, log = process_pages(pages)

for i in range(0, len(page_urls)):

	# Feature vector
	fv = init_fv()

	html = htmls[i]
	page_url = page_urls[i]
	comment = comments[i]
```

```
	log_entry = log_entries[i]

	# Avoid duplicates
	if log_entry not in already_processed:

		# Extract url from comment (all comments at this point have urls)
		comment_url = re.findall(r'https?://[^\s<>"]+|www\.[^\s<>"]+',
					comment[0]

		html = "<head>" + html[1]
		soup = BeautifulSoup(html)
		iframes = soup.findAll('iframe')

		if iframes != None:
			fv['niframes'] = len(iframes)

		scripts = soup.find('script')

		if scripts != None:
			fv['nscripts'] = len(scripts)
			fv['nsuspscript'] = has_susp_script(scripts)

		videos = soup.find('video')
		if videos != None:
			fv['nvideos'] = len(videos)

		if len(iframes) == 1:
			fv['niframes'] = len(iframes)

		fv['ncommsuspterm'] = has_keywords(comment)
		fv['commthastxt'] = has_text(comment, comment_url)
		fv = analyze_iframes(iframes, page_url, fv)
		fv = analyze_url(page_url, fv)

		fv['noccurr'] = log[log_entry]

		filter_pred = labelgen(fv)
		fv['filterpred'] = filter_pred

		already_processed.append(log_entry)
		v = list(fv.values())
		featurevectors.append(v)
		if verbose == 1: print(count, "  ", fv)
		if verbose == 1: print("url:  ", page_url)
		if verbose == 1: print(count, "  ", v)
		if verbose == 1: print("filter pred: ", filter_pred)
		if verbose == 1: print("")
		count += 1

with open(''.join(('../gen/data_', outn)), 'wb') as f:
	np.savetxt(f, featurevectors, fmt='%i')

with open(''.join(('../gen/links_', outn)), 'w') as f:
	for item in already_processed:
		f.write("%s\n" % item)

if verbose == 1: print("\ntotal entries: ", i + 1, " non duplicates: ", count - 1)
end = time.time()
if verbose == 1: print('\ntime elapsed: ', end - start)
```

learn.py

```
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.ensemble import ExtraTreesClassifier
import matplotlib.pyplot as plt

np.set_printoptions(threshold=np.nan)

verbose = 1

filen = 'example_output'
outn = 'example_out'
data = []

# Load labeled training data
labdata = []
with open('../gen/training_lab_data', 'r') as f:
    labdata = f.readlines()

if verbose == 1: print("--- loaded labeled data")

# Remove newlines and separate columns
labdata = [x.strip('\n') for x in labdata]
labdata = [x.split(" ") for x in labdata]

n = len(labdata)
m = len(labdata[0])

xs = [x[: m - 1] for x in labdata]
ys = [x[m - 1] for x in labdata]

if verbose == 1: print("number of 1 (clickjacking): ", ys.count('1'))
if verbose == 1: print("number of 2 (malicious): ", ys.count('2'))
if verbose == 1: print("number of 3 (benign): ", ys.count('3'))

# Create validation set with train-test split 0.5
xs_train, xs_test, ys_train, ys_test = train_test_split(xs, ys, test_size = 0.5,
random_state = 345)

clf = ExtraTreesClassifier(n_estimators=50, max_features = m-1, max_depth=None,
min_samples_split=1, \
       random_state=12, bootstrap=True)

clf.fit(xs_train, ys_train)

prob = clf.predict_proba(xs_test)
pred = clf.predict(xs_test)

# Analyze result
correct = 0
wrong = 0

falseneg = 0
falsepos = 0
# Correctly classified
crtclass = 0
trueclickj = 0
predclickj = 0

for i in range(0, len(xs_test)):
```

```
        result = pred[i]
        if verbose == 1: print(i, " ", ys_test[i], " pred: ", result, prob[i])

        if ys_test[i] == '1':
                trueclickj += 1
        if result == '1':
                predclickj += 1

        if ys_test[i] == '1' and result != '1':
                falseneg += 1
        if ys_test[i] != '1' and result == '1':
                falsepos += 1
        if ys_test[i] == '1' and result == '1':
                crtclass += 1

        if(ys_test[i] == result):
                correct = correct + 1
        else: wrong = wrong + 1

if verbose == 1: print("\n\ntotal: ", n)
if verbose == 1: print("correct: ", correct)
if verbose == 1: print("wrong: ", wrong, "\n")

accuracy = (correct * 100) / len(xs_test)
if verbose == 1: print("accuracy: ", accuracy, "\n\n")

if verbose == 1: print("falsepos: ", falsepos)
if verbose == 1: print("falseneg: ", falseneg)
if verbose == 1: print("crtclass: ", crtclass)

if verbose == 1: print("trueclickjickj: ", trueclickj, ", predclickjickj", predclickj)

importances = clf.feature_importances_
if verbose == 1: print("features as list:\n")
if verbose == 1: print(importances)

# Code from:
# http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html...
# ...example-ensemble-plot-forest-importances-py

# Print the feature ranking
indices = np.argsort(importances)[::-1]
if verbose == 1: print("\nFeature ranking:")
nf = m-1
for f in range(nf):
    if verbose == 1: print("%d. feature %d (%f)" % (f + 1, indices[f],
importances[indices[f]]))

# Plot the feature importances
plt.figure()
plt.title("Feature importance")
plt.bar(range(nf), importances[indices], color="b", align="center")
plt.xticks(range(nf), indices)
plt.xlim([-1, nf])
plt.show()

# Train classifier on full training set
clf.fit(xs,ys)

# Load data
data = []
```

```
with open(''.join(('../gen/data_', filen)), 'r') as f:
    data = f.readlines()
if verbose == 1: print("data file", filen, " loaded")

# Remove newlines and separate columns
data = [x.strip('\n') for x in data]
data = [x.split(" ") for x in data]

n = len(data)
m = len(data[0])

xs = [x[: m - 1] for x in data]
ys = [x[m - 1] for x in data]

out = []
# Print predictions and probability
for i in range(0, len(ys)):
      out.append([i+1, "  pred:", pred[i], " prob:", prob[i]])

with open(''.join(('../out/', outn, ."txt")), 'w') as f:
      for item in out:
            f.write("%s\n" % item)

if verbose == 1: print("output file", outn, " written")
```