Stateless Real Time Cookie Synchronization Detection

Bartosz Struzinski

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh
2022
Abstract

This thesis aims to advance the currently existing methods of cookie synchronization detection by creating and releasing a dataset that can be used to train various machine learning models to perform stateless, real-time cookie synchronization detection. Stateless cookie synchronization detection works by classifying the requests made by the browser, using those requests’ metadata only. Since a stateless method does not consider the requests’ content, it can detect encrypted cookie synchronization, which most existing cookie synchronization detectors cannot do.

The dataset is created by a Selenium-based web crawler that instruments a Chrome browser to visit the top 2000 most popular domains according to Tranco ranking. The HTTP requests made along the way are intercepted, labelled by a heuristic cookie synchronization detector, and parsed to create a dataset of 129,515 data points. Out of those, 16,061 (8.06%) are observed cookie synchronization events.

The assembled dataset, containing categorical variables, is then encoded into numerical data so that selected machine learning classifiers can be trained on it. This thesis considers several encoding schemes, such as mean target and frequency encoding, as the choice of the encoder can significantly affect the model performance. Support Vector Machine model with mean target encoding proved to be the best performing combination, achieving a large weighted F1 and AUC scores of 0.963 and 0.959, respectively.

Finally, a decision tree classifier is chosen from the trained models and used to create a proof-of-concept Chrome plugin, performing cookie synchronization detection in the browser. Thanks to its simplicity and easily interpretable graphical user interface, the extension can be used by users with no technical background to observe the trackers performing cookie synchronization in real-time.
Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Bartosz Struzinski)
First and foremost, I would like to thank my supervisor, Prof. Kami Vaniea, for her continuous support during this Master’s Thesis. Her experience and knowledge helped me formulate my research goals, navigate technical and academic obstacles, and motivate me at every step of this project.

Second, I would like to express my most profound appreciation to my Mother, whose unwavering support and candid comments have always inspired me to raise the bar higher, and to my Father, whose hard-working personality and unparalleled conscientiousness will always drive me to excel at whatever I do. I will never be able to express my gratitude for the countless times they helped me throughout my journey at the university and beyond. Mom and Dad, I hope I am making you proud.

Finally, I want to thank my Grandparents for their wisdom and unconditional love. I am convinced that if it weren’t for their mature perspective on life, I would have never pursued a Master’s Degree.
# Table of Contents

1 Introduction .......................................... 1

2 Background ............................................. 4
   2.1 Cookie Synchronization .............................. 4
   2.2 Previous Work Carried Out ......................... 5
   2.3 Related Research ................................... 6
      2.3.1 Privacy Measurement Platforms ................. 6
      2.3.2 Cookie Synchronization ......................... 7
   2.4 Contributions ..................................... 7
   2.5 Ethical Considerations ............................. 8

3 Data Collection ....................................... 9
   3.1 Logging browsing data .............................. 10
   3.2 Classifying HTTP Requests ......................... 13
   3.3 Creating The Dataset ............................... 14
      3.3.1 Selecting Features ............................ 14

4 Training The Classifiers ............................. 18
   4.1 Dealing With Imbalanced Classes .................. 18
   4.2 Feature Encoding .................................. 19
   4.3 Logistic Regression ................................ 21
   4.4 SVM .................................................. 22
   4.5 Decision Trees ..................................... 24
   4.6 Choosing The Right Classifier ..................... 26
   4.7 Removing Redundant Features ....................... 27

5 Proof-Of-Concept Plugin ............................. 30
   5.1 Requirements ....................................... 30
   5.2 Design ............................................. 31
   5.3 Implementation .................................... 33
      5.3.1 Client .......................................... 33
      5.3.2 Server .......................................... 35

6 Discussion ........................................... 37

7 Conclusions ........................................... 39
Chapter 1

Introduction

SEN. WICKER (MS) : There have been reports that Facebook can track a user’s Internet browsing activity, even after that user has logged off of the Facebook platform. Can you confirm whether or not this is true?

ZUCKERBERG : I know that the—people use cookies on the Internet, and that you can probably correlate activity between—between sessions.

We do that for a number of reasons, ... including measuring ads to make sure that the ad experiences are the most effective, which, of course, people can opt out of.

[1]

CAN they really? The above excerpt from the Facebook chief executive’s hearing before the US Senate’s Commerce and Judiciary committees shows that users’ privacy on the Internet has been a longstanding subject of public and political debate. And for a good reason. As the pool of web users grows, the economic and political incentives encourage many parties to base their entire business models on collecting, analyzing and sharing user data. These motives have spawned sophisticated tracing technologies, which are difficult to remove or even detect for users with little to no technical background. HTTP Cookies, or cookies for short, are one of the web technologies adopted for user tracking.

To address the issue of online privacy, researchers and developers have been creating various tools suited for different purposes. Ghostery, a free and open-source privacy browser extension, is an example of a defensive tool, empowering its users to block ads and stop trackers. OpenWPM, created at Princeton University, is an analytical software that has been adopted as a state-of-the-art web privacy measurement framework since its creation. Finally, Mozilla Lightbeam is a simple informational tool available as a Firefox extension. It allows users with no technical background to learn about domains tracking their online activities.

However, none of these tools specifically addresses the issue of cookie synchronization, which is the practice of online trackers sharing data on individuals they track. As defined by Alan Westin, the father of modern privacy laws, privacy is defined as "the claim of individuals (...) to determine for themselves when, how, and to what extent information about them is communicated to others" [2]. Following that definition,
cookie synchronization is an incredibly privacy-intrusive mechanism by design. After all, it is nothing but personal user identifiers being communicated among trackers to improve user profiling. Since all that happens without users’ knowledge, it is paramount to augment the existing tools, such as Lightbeam, with the ability to detect and report cookie synchronization.

Such a feature is of interest not only to everyday web users who have every right to know which domains collect and share their data. It is also essential for policymakers. Some of the related studies, such as [3], indicate that cookie synchronization usage has increased after the advent of GDPR. Thus, tools accurately detecting cookie synchronization are of interest not only to users who care about their privacy but also to policymakers so that the effectiveness of the introduced legislation can be assessed, as well as to researchers studying the effect of directives, such as GDPR, on the landscape of online privacy.

Some of the existing privacy tools already allow for the detection of cookie synchronization. Implementation of an algorithm performing such detection was described in the first year of this project. However, almost all currently existing solutions have one major flaw - they cannot detect encrypted cookie synchronization, which has been reported to be used by high profile trackers such as Google’s DoubleClick. Those studies that can do that ([4], [5]) are either by design unusable within a browser (making them unsuitable for everyday users) or are proprietary software hidden from the public.

This thesis advances the capabilities of open-source privacy-related projects by creating and releasing a cookie synchronization dataset suitable for training ML classifiers. As such a dataset has never been created before and publicly released, doing so can enable and encourage future researchers and developers to invent more advanced cookie synchronization detectors or privacy tools in general. Once we create the dataset, various machine learning models are trained to identify HTTP requests that, based on their metadata, are most likely cookie synchronizations. These content-agnostic, or stateless methods do not consider the actual content of classified requests. They thus can identify the aforementioned encrypted cookie synchronization, allowing for a more reliable measurement of how prevalent data sharing among the trackers is. Unlike the approach in this thesis, most previous work does not deal with encrypted cookie synchronization and hence can only report lower bounds on the frequency of this privacy-intrusive mechanism.

Finally, the machine learning model created in this thesis is used to implement a proof-of-concept browser plugin detecting cookie synchronization in real-time and reporting its findings to the user. This plugin aims to resolve the issue of currently available tools being too challenging to use by users with no technical background. Although the extension is created for everyday browser users, it can be considered a minimal viable product of a tool targeted at policymakers and privacy researchers. Thus, the stakeholders of this thesis include all of the groups mentioned above.

Figure 1.1 presents the plugin created in this project. The displayed graph is a visualization of cookie synchronization performed by tracking domains, in this case during visits to just three domains: cnn.com, bbc.com and twitter.com. That graph has been created by classifying HTTP requests made by the browser. Nodes represent domains engaging in cookie synchronization, and edges represent detected instances of cookie
Several aspects of the above plugins are worth noting. First of all, it is the first known plugin capable of detecting and reporting cookie synchronization. Unlike the existing plugins, such as the aforementioned Mozilla Lightbeam, the extension created in this thesis tells its users which parties share their data and not just collect it, giving an insight into a whole different aspect of user privacy. Second, the method used by the plugin to detect collaborating trackers is based entirely on machine learning, again differentiating it from all publicly available cookie synchronization detectors.

The performance of the ML model created in this thesis has been proved to be very good - the final decision tree classifier achieves high F1 and AUC scores of 0.981 and 0.956, respectively. The only other study using machine learning for cookie synchronization detection (Papadopoulos et al. [5]) reports a slightly lower F1 and AUC scores of 0.898 and 0.956, respectively. However, since their study was performed on a different, undisclosed dataset, one must be careful with directly comparing both classifiers. Similarities in achieved scores should be treated as sanity checks proving the correctness of design choices in this project.

Ultimately, the goal of this thesis is to design an end-to-end methodology for creating a privacy plugin, focusing on discovering cookie synchronization in the wild. Each step, from assembling a cookie synchronization dataset from scratch, creating an ML model, and developing usable privacy-related software, is discussed in great detail in the following chapters.
Chapter 2

Background

This chapter introduces the concepts required for understanding the project. It begins with a description of plaintext and encrypted cookie synchronization. The previous work carried out as part of last year’s project is then outlined, along with an overview of related research. Finally, the ethical considerations of this project are considered.

2.1 Cookie Synchronization

**Plaintext Cookie Synchronization** An important concept relevant to HTTP cookies is the *Same Origin Policy*. This security mechanism allows a domain to access an HTTP cookie only if that domain created it. *Cookie Synchronization* is a process used to bypass that policy. It relies on tracking domains sharing pseudonymous identifiers, stored in cookies and associated with a specific user, amongst each other. Cookie Synchronization is a way for domains to share cookie values and improve user profiling [6]. It is primarily a three-step process.

1. A user visits two host domains, website1.com and website2.com. If third parties, say tracker1.com and tracker2.com, are present on these websites, they assign the user two identifiers, id123 and idABC respectively.

2. A user makes another direct visit to website3.com, on which tracker1.com is present, but tracker2.com is not. Resources from tracker1.com are loaded into website3.com Document Object Model.

3. The request loading resources from tracker1.com is then redirected to that tracker's partner, tracker2.com. Alternatively, a script loaded from tracker1.com makes a separate request. Either way, the request made to tracker2.com contains the unique ID that tracker1.com assigned to the user, embedded somewhere in the request, for example as a URL parameter. Assuming that tracker2.com has already assigned the user an identifier idABC, along with the redirected/new request tracker2.com receives a cookie storing that id. Now, tracker2.com can easily pair its identifier with those of tracker1.com.
Encrypted Cookie Synchronization  The above paragraphs explain vanilla cookie synchronization. However, as observed by previous studies in that domain, some ad exchanges, such as Google’s DoubleClick, create and use a one-way hash of the cookie as user id ([7], [6]). One reason for that is to protect the actual cookie from being revealed to unwanted parties that may continuously observe the user’s traffic, such as browser extensions or even Internet Service Providers [5].

The second reason for cookie synchronization encryption is to protect the real cookie from being revealed to partnering trackers [4]. Under the plain-text assumption, if two trackers synchronize a cookie with a particular source domain, both will receive the same cookie value set in the user’s browser by that source domain. Once that is done, these two third party domains are free to synchronize their IDs with each other, to realize that they hold information about the same user [5]. As such a scenario goes beyond what the source domain would typically intend to achieve, cookie encryption was implemented. With that approach, both third parties would receive a different user id and hence be unable to learn that they are observing the same user.

From a privacy researcher’s perspective, such an approach to cookie synchronization has a rather unwanted consequence - it is more difficult to detect it. This observation acts as the motivation for a content-agnostic cookie synchronization method, which is one of the focuses of this study.

2.2 Previous Work Carried Out

Last year’s part of this project measured how user interaction with GDPR privacy warnings affects the ecosystem of third parties tracking that individual’s activity. To that end, a privacy research framework was created. Cookie Crumble Tracer, or CCT for short, is a Selenium-based crawler capable of interacting with GDPR privacy warnings to either agree or disagree to analytical cookies. Thus, the platform can measure how the user’s privacy choices affect the characteristics of cookies saved in the browser, as well as the communication in form of cookie synchronization between the trackers.

The collected information was then used to create the network graphs representing the tracking environment. Graph analysis with various centrality metrics showed that while opting out from accepting analytics cookies reduces the number of observed trackers and the amount of data shared among them, it poses no threat to the dominance of well-known trackers, such as Google or Facebook.

The main contributions of last year’s project were the implementation of a cookie synchronization detection algorithm and creating a GDPR privacy measurement framework. However, the created algorithm was unable to detect the encrypted variant of cookie synchronization. The framework, aimed at increasing privacy awareness among technical and non-technical users, was difficult to use and required specialized knowledge to set up. This year’s study works on both of the issues by focusing on creating a stateless (content-agnostic), machine learning classifier for cookie synchronization detection. What is more, that model is later used to create a proof-of-concept browser plugin for real-time cookie synchronization detection, which is easy to use by non-technical users.
2.3 Related Research

2.3.1 Privacy Measurement Platforms

One of the most important privacy measurement platforms created to date is OpenWPM [8]. This tool, aimed at enabling flexible and modular web measurement, includes novel methods for detecting persistent tracking techniques, such as device fingerprinting and cookie synchronization. As OpenWPM can be expanded through browser extensions to include custom instrumentation, it has been adopted as a state-of-the-art measurement platform in web privacy. The shortcoming of OpenWPM is the same as in all, but one currently existing research platforms - its cookie synchronization detection mechanism is purely heuristic. It relies on simple string matching to detect id sharing, making it unable to detect encrypted cookie synchronization.

The only platform created to date dealing with encrypted cookie synchronization is CONRAD, or COokie syNchRonizAtion Detector, created by Papadopoulos et al. [5]. It is a holistic mechanism allowing for real-time cookie synchronization, even if synchronized IDs are encrypted. CONRAD uses a decision-tree-based classifier trained on 179 million HTTP requests collected from 850 volunteers over 12 months to detect cookie synchronization events. To label the dataset, a heuristic cookie synchronization detection algorithm, similar to the one shown in Figure 3.2 was used. Features considered by the classifier have been constrained to those available during the user’s web browsing so that the requests can be classified in real-time. As far as model performance is concerned, CONRAD achieves an F1 score of 0.898, an AUC metric of 0.946, and manages to identify 3.771% more cookie synchronization cases than other related studies [5].

Mozilla Lightbeam is another popular privacy measurement tool. Lightbeam differentiates between functional and tracking cookies by analyzing cookies stored in the browser. The plugin then displays a graph of sites visited by the user, along with cookies placed, allowing the user to learn which parties are capable of tracking them across different sites. Although Lightbeam does not detect cookie synchronization, it serves as one of the most important motivations for this thesis. It showcases the need for plugins aimed at a mainstream audience, producing real-time visualization of user tracking. According to Till Faida, the co-founder of Adblock, Mozilla Lightbeam ”represents a step forward in the fight for greater openness across the internet” [9].

The introduction of Mozilla Lightbeam signified a critical shift within the open-source community. However, a turn towards better privacy and the need for privacy awareness is also visible within the commercial environment. An update to Apple’s Safari browser, included in iOS 14, gives the user the ability to learn about the trackers currently profiling them and the websites those trackers have been contacted by. Although how that privacy report is generated remains a mystery, one could assume that the method is based on the analysis of third-party cookies stored in the browser, as after deleting them, the report becomes empty.
Chapter 2. Background

2.3.2 Cookie Synchronization

The first large-scale study of cookie synchronization was conducted by Acar et al. Their heuristic cookie synchronization detection has been used by almost all subsequent studies focusing on this mechanism and is the basis for the algorithm utilized in the previous year of this project. In their study, Acar et al. showed that cookie syncing could greatly amplify privacy breaches through server-to-server communication between trackers - by utilizing this mechanism, the number of trackers capable of reconstructing at least 40% of a user’s browsing history increases from just 0.3% to 22.1% [10].

Olenik et al. researched cookie synchronization to quantify the amount of data flow between parties in a process known as Real-Time Bidding - a novel paradigm (involving cookie synchronization) of serving curated ads, in which buyers bid in an auction to serve an ad to the user. The study concluded that 91% of investigated users were affected by cookie synchronization and in some cases, up to 27% of their browsing history could be leaked to third parties [7]. Although Olejnik et al. were the first to point out the shortcoming of heuristic cookie synchronization algorithms, their approach still utilizes this somewhat flawed approach.

The importance of detecting cookie synchronization is showcased by Papadogiannakis et al., who report that on websites on which third parties perform cookie syncing, each id is shared on average 3.51 times, even before users accept or deny consent. If a user opts out, that number increases to 3.91 shares per id to make things worse. Finally, their study also reports that the more popular a website is, the more likely it is to disregard users’ consent and engage in cookie synchronization with other parties [3].

2.4 Contributions

This study takes inspiration from many of the aforementioned studies. Similarly to Papadopoulos et al. [5], a classifier capable of stateless, or content-agnostic, real-time cookie synchronization detection is created. However, unlike that particular study, this project is not limited to decision trees - logistic regression and support vector machines models are also considered. The trained models also consider features that have not been explored by Papadopoulos et al., as described in the section 3.3.1.

Unlike in the case of [5], one of the main contributions of this study is a dataset release. Creating and releasing a cookie synchronization dataset on which machine learning models can be trained has never been done before and is thus meant to encourage and advance the possibilities and ways of detecting cookie synchronization. Since this report, in particular Chapter 3, serves as the required documentation, the dataset will be released through university channels only after the project submission.

This thesis additionally proposes a design and a proof-of-concept implementation of a browser extension that would allow users to get a real-time classification of their HTTP traffic in a visual privacy report. Although there are platforms to detect cookie synchronization, such as [8], they do not deal with obfuscated cookie synchronization requests. CONRAD, the only platform tackling this issue, aims at mobile environments rather than desktop browsers and has not been made public by its creators. Moreover,
these platforms are complicated to use, as they often involve setting proxies, making them unusable for non-technical users.

The contributions of this paper in the form of the dataset, a machine learning classifier and a proof-of-concept browser plugin can be transformed into auditing tools for policymakers and regulators for issuing more reliable privacy grades to the domains we all visit. Furthermore, the ultimately created tool can be seen as an awareness increasing platform aimed at non-tech-savvy internet users. One can consider it a possible improvement over plugins such as Mozilla Lightbeam, which only shows trackers present on the websites a user visits. The extension created in this thesis, on the other hand, shows how those trackers communicate, thus giving the aforementioned potential shareholders an insight into different privacy aspects.

### 2.5 Ethical Considerations

As all goals of this study, from creating and releasing a cookie synchronization dataset to the design and implementation of a proof-of-concept plugin, are within the domain of online privacy, it is only appropriate for this project to promote and adhere to ethical standards of data collection.

A web crawler is created to collect HTTP traffic logs from which the released dataset is assembled. This bot exhibits entirely passive web behaviour, which involves only visiting websites available to the general public and investigating the traffic incoming and leaving the browser. As this limited browsing pattern, considered standard user behaviour, presents no harm to the websites’ publishers and does not perform any indexing, the crawler does not adhere to the specifications of `robots.txt` files.

According to Vallor et al., good data collection practices involve storing only as much data as one needs, for only as long as needed to accomplish given task [11]. Following this principle, once the initial dataset is created, this thesis will also focus on identifying those features that are irrelevant to predicting cookie synchronization and thus can be discarded in this and future work. Although this might seem like a redundant task in the case of this study, as no personal data is collected, one has to keep in mind that one of the goals is to create a plugin that could be commonly used. In that case, sensitive data would undoubtedly be collected to classify HTTP requests, making it is only appropriate to apply good data collection practices, starting with this thesis.
Chapter 3
Data Collection

The first contribution of this study is creating and releasing a dataset that can be used to train a machine learning model classifying HTTP requests as cookie synchronization.

The process of collecting data for the aforementioned dataset requires a log file of all requests, with their content, made by the browser. Those requests must be labeled, for example by a deterministic cookie synchronization algorithm, such as one described by [5], which tell us which requests were observed to be cookie synchronization. There are two ways of obtaining the traffic logs.

The first one requires a group of study participants that would either volunteer or be incentivised to share their browsing data for the study. The obvious strength of this approach, particularly if maintained over a long period and many web users, is that the dataset reflects real browsing patterns and is therefore reflective of the real world tracking landscape. Since browsing history is sensitive data, traffic logs must be properly anonymized, which in this case is not a matter of simply removing some kind of user id. Logs of HTTP requests contain many different, potentially sensitive, types of information (geolocation, session-ids, cookies) which can be carried in various parts of the requests, such as headers, URL parameters or body. Of course, another thing to consider when collecting real user data is how that data is going to be retrieved from their browsers. Papadopoulos et al. collected their data by setting a group of proxies fronted by a load balancer. Furthermore, their study participants were provided with free data plans as long as they are using the proxies.

Conducting a study on real users requires a large overhead which is not attainable from the perspective of this thesis. Therefore, in this project a more simplistic approach is taken, in which data is collected by a web scraper. It is paramount that this scraper emulates human-like activity. Otherwise, if the webserver to which a request is made detects non-human behaviour, for example by obsolete User-Agent header, it may respond with content heavily modified relative to what a normal user would see. Englehardt and Narayanan have previously compared the content served to full-fledged Chromium-based browsers and stripped-down ones, such as PhantomJS. According to their study, PhantomJS loads around 30% less HTML files and 50% plain-text files [12]. Moreover, many servers do not serve ads to stripped-down browsers, which is
particularly important from the perspective of our study - it is during serving the ads and Real-Time Bidding when a large portion of cookie synchronization events is expected to occur.

To emulate human-like behaviour this project utilizes Selenium, a lightweight framework for functional web app testing which supports automating browser activity and allows for interaction with and manipulation of the Document Object Model [13]. Although Selenium supports the automation of all major web browsers, this study will use it exclusively with Google Chrome, as it is currently the most popular web browser in existence [14] and allows third party cookies without blocking any.

### 3.1 Logging browsing data

To create a dataset on which machine learning models can be trained, HTTP data has to be obtained, which is the purpose of the aforementioned Selenium web crawler. The number of requests our web crawler observes becomes a significant consideration, as it directly specifies the size of the created dataset. Although HTTP requests have a specific structure, their metadata (e.g. headers) can take virtually unconstrained values. Thus, to develop meaningful insights about the nature of classified HTTP requests, a dataset of large size and containing requests from various domains is desired. For this reason, the web crawler collects the data on visits to the front pages of 2000 of the most popular domains as specified by Tranco ranking [15], which is four-times as many websites as used in the first year of this project.

Another approach to creating a large dataset is to focus on fewer domains, but perform deeper crawls, beyond that website’s landing page. The advantage of such an approach is that eventually, the domains tracking the crawler would serve ads that are more relevant to its synthetic activities. This, however, is of no value to this project. Crawling more websites, on the other hand, would increase the number of domains that could save a cookie tracking the activity of the bot, and later leak it to their partnering third parties, thus also increasing the amount of cookie synchronization.

To capture HTTP requests one would need a proxy server acting as an intermediary between client requests and server responses. Normally this would be done by using proxy servers and packet analyzers such as Fiddler or Wireshark. However, as these tools lack integrated API or command-line interface, they are virtually useless in the case of an automated web crawler. As a solution, Selenium’s Wire extension is used. Wire bundles core Selenium framework with a proxy server accessible via API, thus enabling the interception of the underlying requests to and from our browser.

Selenium web crawler gathers HTTP requests by instrumenting Chrome browser, running in a separate Chrome profile so that other web activity on the machine doesn’t interfere with the web crawler and to make the browser’s state persistent between the crawls. To increase the crawler’s fault tolerance, it has been designed not to visit all 2000 Tranco domains in one crawl. Instead, the data collecting bot crawls 500 domains at a time (or any user-specified number), periodically saving the result to the disk and only then performing another crawl over the next 500 domains. In this way, if for some reason a crawl fails, the bot is not forced to resume the data collection from scratch.
Assigning the crawler with a separate Chrome profile allows the cookies to persist and be reused by trackers in subsequent crawls. Using Selenium without a dedicated Chrome profile would result in the crawls being run in temporary profiles, thus not keeping the cookies between the crawls and significantly decreasing the amount of observed tracking.

Figure 3.1: The process of collecting HTTP data by Selenium-based web crawler.
Chapter 3. Data Collection

Figure 3.1 presents a diagram for collecting traffic data by the web crawler. For each of the 2000 domains, the crawler uses Selenium to instruct the Chrome browser to visit it and agree to analytics cookies, which aims at maximizing the amount of observed user tracking. The cookies are accepted by identifying elements within GDPR cookie banners that, when clicked, express consent to all forms of user profiling. The heuristic process of identifying those elements is the same as in the first year of this project, and therefore will not be fully described in this part. In short, the crawler uses Selenium to access the Document Object Model of the visited websites and extract those elements that could be clicked by the user. Then, the crawler uses a list of phrases commonly used to express user consent, created by Molnar [16], to identify the clickable element responsible for opting into tracking. In this way, the amount of cookie synchronization requests in the dataset is maximized.

Once the crawler accepts to being tracked, it uses Selenium Wire to extract the requests made on the visit to the current domain. Once extracted, these requests are cleared from Selenium’s memory. This ensures that each time a new website is visited, the crawler extracts only the requests made when visiting that particular domain. Otherwise, without clearing Selenium’s memory, the crawler would extract all requests made by the browser up to that visit. This is not desired - since the domain being visited is one of the features considered by the classifier, the crawler must know on which domain a given request was made.

The last two things that the crawler does before exiting are extracting the cookies stored in the browser and parsing them for efficient storage and retrieval. The cookies are extracted from the browser using the Chrome DevTools protocol. In particular, the crawler uses Selenium to execute the Network.getAllCookies within Chrome, which returns all cookies stored in the browser. The retrieved cookies, together with HTTP requests already captured, are used in the subsequent steps to classify logged traffic as cookie synchronization or not.

Once the cookies are extracted from the browser, they need to be filtered so that all non-tracking cookies are removed from the set. This is done by following the approach taken by the previous research in the privacy area.

First of all, all short-lived cookies are removed. According to Englehardt et al. [17], a tracking cookie must have a lifespan of at least three months, so that it can track a user’s activity over a substantial period. Hence, following that approach, all cookies with an expiration date lesser than 90 days from the day of creation are removed.

According to [18] and [10], each cookie’s value field can be a string holding multiple name-value pairs, each of which can be a user identifier. Thus, the second step is to split the string within each cookie’s value field on delimiter character matching a regular expression [ˆa-zA-Z0-9 =-]. In this way, a list of separate, long-lived user identifiers is obtained, along with domains that have set them in the user’s browser.

Third, an information-theoretical approach of measuring cookies’ entropy to further prune the cookies dataset is taken. The assumption is that a user-tracking cookie should have enough entropy to uniquely identify an individual from a large group of internet
users. Following the method illustrated by [19], zxcvbn score is used \(^1\) to reject those cookie identifiers which do not have the score of at least \(10^9\).

As the last step, inspiration from [18] is again taken, by first removing identifiers shorter than 7 characters and then removing those name-value identifiers that have been observed to be inconsistent. A name-value identifier is inconsistent if, for a particular domain, there are multiple identifiers with the same name but different values. For example, name-value identifiers (id, sg29dj9wj) and (id, ks62bg6pe) set by facebook.com would be excluded - for a particular user a tracking cookie set by a particular domain would have a single consistent value.

Another filter that could be applied to the remaining cookies is excluding those identifiers that are shared among different users. For example, if two distinct users receive the same name-value identifier, say (id, sg29dj9wj), from facebook.com, this inevitably means that this particular identifier could not be a tracking cookie - if it was, distinct users would have distinct cookie values. Unfortunately, as this study is a single user or rather, a single bot, project, such an approach is not attainable. Although a multi-user approach could be emulated by running two distinct crawls in separate Chrome plugins, this improvement is left to the future work as it would significantly prolong the duration of data collection by the crawler.

### 3.2 Classifying HTTP Requests

At this point the crawler has produced:

- a log file of HTTP requests made by the browser while visiting the top 2000 domains from the Tranco list
- a parsed list of cookies extracted from the browser which are potentially tracking identifiers, together with the domains that set them.

These resources make it possible to label the logged requests using a heuristic algorithm cookie synchronization algorithm. The working of the algorithm implemented for that purpose, previously utilized by [10], [21], [22], is presented in Figure 3.2.

As cookie synchronization is simply a request to a third party carrying at least one tracking cookie set by another domain, a heuristic cookie synchronization detection algorithm looks at the contents of the request, finds all suspicious strings that could potentially be a user identifier, and searches the list of cookies that have been extracted from the browser to determine if any of those suspicious strings has been observed to be a cookie set by a different domain. In particular, each request’s URL, Cookie header, Referrer header and body are searched for the suspicious strings, an approach previously by Papadopoulos et al.

To decide if two domains are the same, the approach taken by [5] is followed, by utilizing DNS whois protocol to obtain information about the organization owning a particular domain. This approach can discriminate between intentional cookie synchronization

\(^1\)zxcvbn is a score measuring password strength. It defines the amount of information it carries by calculating the number of guesses it would take for an adversary to learn it [20].
and legitimate information passing within an organization. A simplified approach involving ordinary string comparison would not be able to tell that cookies set by store.steampowered.com and steampowered.com belong in fact to the same domain, which is why a more complex approach is used.

### 3.3 Creating The Dataset

With the above algorithm in place, each request in the logs of HTTP traffic captured by the web crawler is labelled with boolean values indicating either a cookie synchronization request (True label) or normal HTTP request (False label), allowing for parsing the entire collection of labelled network traffic into the final dataset that will be used for model training.

#### 3.3.1 Selecting Features

One of the contributions of this study is to produce a dataset that can be used to create a classifier recognizing cookie synchronization events in real time, as the user browses the web. Thus, one of the constraints on features that the dataset will contain is that all of them must be available at run time.

Papadopoulos et al. [5] conducted the only study to date which investigated machine learning techniques as means of real time cookie synchronization. In their study, the following features were extracted from each HTTP request to train the classifier:

- EntityName: the domain to which the request is made
- TypeOfEntity: category of the EntityName, restricted to Content, Social, Advertising, Analytics, Other
Chapter 3. Data Collection

- **ParamName:** \{aid, u, guidm, subuid, tuid, etc.\}
- **WhereFound:** \{parameter in URL, parameter in Referrer, in the URL path\}
- **StatusCode:** HTTP response code, eg. 200
- **Browser:** \{Firefox, Chrome, Internet Explorer, etc.\}
- **NoOfParams:** number of GET parameters in the URL

The underlined features are the ones which are either questionable or have been ambiguously described. From the context of [5] it is unclear what features ParamName and WhereFound mean, although one can assume they denote the name of the identifier being shared. However, it was observed in the previous year of this thesis that sometimes only the value part of the cookie is shared, without its name. In this case it becomes unclear what the value of ParamName should be. Moreover, some HTTP requests have been observed to carry shared identifiers in many parts of the request, which again makes it unclear what the value of WhereFound should be. Finally, [5] provides no justification for using the type of browser as another feature considered by the classifier. A possible motivation for including that information is that some of the modern browsers, such as Firefox, include integrated functionalities to block trackers that follow users online and collect information about their browsing habits and interests [23].

For the reasons mentioned above it has been decided not to use those features, but rather take inspiration from them to come up with the following ones:

- **Referer:** HTTP request referer, which is the address of the page that makes the request. If that value is for some reason unavailable, the request initiator, which is the website being visited, is used instead.

- **Referer\_category:** category of the referer domain, eg. Social Networking. Unlike in the case of [5], multiple available resources are used to classify the domains into 50 distinct categories, plus Unknown, instead of just 5. The methodology of collecting this information is described in the next section.

- **Netloc:** the domain to which the request is made, extracted from the request URL.
- **Netloc\_category:** category of the netloc domain, similar to Referer\_category.
- **Response:** HTTP response code.
- **Sus\_url:** true if a suspicious string \(^2\) was observed as a GET parameter in the URL, false otherwise.
- **Sus\_headers:** true if a suspicious string was observed within the request’s headers, false otherwise.
- **Sus\_headers:** true if a suspicious string was observed within the request’s body, false otherwise.
- **Redirect:** true if the request was a result of a redirect, false otherwise.

\(^2\)A string is considered suspicious if it has at least 7 characters and zxcvbn score of at least 10\(^9\).
• Cache_response: the value of Cache-Control response header or default-cache-control if no such response header is found.

• Sec_fetch_site: the value of Sec-Fetch-Site request header or default-sec-fetch-site if no such request header is found.

• Sec_fetch_dest: the value of Sec-Fetch-Dest request header or default-sec-fetch-dest if no such request header is found.

• Sus_param_shared: true if the HTTP request contains a cookie name that has been observed to be shared during cookie synchronization, false otherwise.

• Label: boolean value indicating whether the request was a cookie synchronization event.

As one can see, it has been decided to include significantly more features than in the case of [5]. In addition to the the domain to which the request is made, and its category, the referring domain and its category is also considered. After all, one can argue that a HTTP request referred by a well known tracking domain is more likely to be a request leaking one’s identifier than, let’s say, an educational website.

Furthermore, instead of one feature such as WhereFound, the approach in this thesis considers three separate boolean values Sus_url, Sus_headers and Sus_body, each one denoting whether a potentially tracking string was observed. This approach allows the classifier to consider those requests which carry suspicious string in more than one part. Finally, a new feature Sus_param_shared is created, telling if any part of the request contains a parameter name previously observed to be a name of a synchronized cookie.

The approach taken in this project also considers whether the request was originated by a redirect. After all, as described in Section 2.1, cookie synchronization is often initiated by a redirect issued by one tracker to its collaborator. What is more, some more sophisticated request metadata indicating the origin and purpose of the request is also considered in the dataset. Sec_fetch_dest is a request header indicating the request destination, which is where and how the data fetched by the request will be used. For example a script value indicates that the destination is a script, and the request might have been initiated from an HTML script tag [24]. Considering that one of the ways of initializing cookie synchronization, apart from redirecting a HTTP request, is issuing a separate request from an embedded script, this particular header may indeed prove useful in differentiating cookie synchronization requests from the normal ones. Also considered is Sec_fetch_site feature which is another request header indicating the relationship between a request initiator’s origin and the origin of the requested resource [25]. For example, if the request initiator and the server to which request is made have a different site, eg. a request by trackerA.com to trackerB.com, that header should be set to cross-site. Since by design all cookie synchronization requests are made between different domains, it has been decided to include this piece of information as well.

Finally, the response Cache-Control header is also taken into account because, unlike images or text files, cookie synchronization requests should not be cached. The reasoning behind this hypothesis is that, when targeting a user, the trackers would want to exchange the most up-to-date information. Moreover, as cookie synchronization is
used during serving Real Time Bidding, one could assume that cookie synchronization should be repeated every time a user is served with retargeted ads.

**Domain Classification**

Generally speaking, categories of the domains participating in cookie synchronization cannot always be inferred from HTTP request alone. To obtain this information, other publicly available resources must be utilized.

This thesis follows the approach taken by Althobaiti et al. [26] - in particular, FortiGuard website categorization service [27] is used to obtain categories of all domains in the dataset. However, FortiGuard does not provide any API, but a simple form instead. For obvious reasons, manual classification of more than 4000 domains is unfeasible. Therefore another, single-purpose web crawler is created. This bot’s only task is emulating human behaviour and using the aforementioned form to classify every domain in the dataset.

The crawler takes as an input a CSS selector that can be used to uniquely identify both the form and the submit button on FortiGuard’s website. With these resources, the crawler extracts all distinct domains from the dataset and inputs each domain into the form. After submitting the form, the scraper waits for at most five seconds and retrieves the category of the website. If FortiGuard fails to classify a domain, it is saved to be later reviewed by a human actor and categorized manually.

Once all domains are categorized, the crawler saves the domains with corresponding categories in a PostgreSQL database so that it can be joined with the relation holding the aforementioned HTTP features and produce the final dataset on which classifiers are trained.
Once a labelled dataset has been created, various machine learning algorithms can be used to classify HTTP requests, from Naive Bayes to even more advanced methods such as Multilayer Perceptron. However, as ultimately, the classifier is supposed to be used for real-time cookie synchronization detection, a middle ground between the computation cost for training and good accuracy must be found.

This chapter describes and contrasts three different approaches to classifying the requests. It also discusses the techniques to handle aspects such as categorical values and the imbalance between classes in the dataset.

4.1 Dealing With Imbalanced Classes

The created dataset contains data points, one for every web crawler’s HTTP request. However, as one might expect, most of these requests are not privacy-intrusive events but functional requests delivering actual user content. Figure 4.1 shows how imbalanced the created dataset is. Out of all data points, 113,454 are regular requests, while only 16,061 (8.06%) have been identified as id sharing requests.

In such a situation, the accuracy paradox comes in the way and makes this metric unusable for the classification task - by simply predicting class 0 (not a cookie synchronization) all the time, one could easily reach an accuracy of 91.94%. This metric therefore cannot be used to evaluate the performance of our classifiers. Instead, metrics such as precision, recall, and F1 score will be used to evaluate the models.

These metrics allow for a more reliable evaluation of trained models. However, it is still not enough to overcome the problem of imbalanced classes during the actual model training. For example, when training a logistic regression model, the average of the log loss across all samples in the dataset is calculated. Based on the reported cost value, weights corresponding to particular features are updated, resulting in slightly different predictions in the next iteration. Unfortunately, as an average of losses across all samples is used, the algorithm could still learn to correctly predict mostly the majority class, achieving relatively high accuracy.
A method for “levelling” the costs associated with the classes is thus needed. One approach is to resample both classes, either using down-sampling or over-sampling. A rule of thumb is to use under-sampling when the dataset is large enough, and one can discard some data points, or over-sampling if the dataset is small and each sample is valuable. Although the only study relevant to our project ([5]) does not specify the technique used, it is known that their dataset is significantly larger than the one created in the previous chapter. Thus, the dataset created in this thesis should be considered relatively small, hinting at the suitability of over-sampling. As an alternative to under or oversampling, one could try and generate synthetic samples from the minority class.

With no prior knowledge about the HTTP request features and their labels, resampling is changed for a simpler, more robust and logical approach. Since, from the perspective of this thesis, the requests labelled as cookie synchronization are more valuable than standard requests (after all, these are the requests that the model is trying to identify), both classes can be assigned a weight representing their values. In practice, each class is assigned a weight that is inversely proportional to the number of samples belonging to that class. The ratio of True Class to False Class is 1:7.06. Thus, True Class is assigned a weight of 7.06, while the weight of False Class remains 1. In this way, samples representing cookie synchronization contribute to the overall loss more.

### 4.2 Feature Encoding

The created dataset contains 13 features, all with categorical data, which at this point makes it unusable for training classifiers such as Logistic Regression or Support Vector Machine. Thus, the features must first be encoded to turn their category levels into numerical values. However, this must be done carefully as many algorithms’ performance may vary significantly depending on the feature encoding [28]. In the case of this project’s dataset, four different encoding schemes are discussed: one-hot dummy, ordinal (also known as label), mean target and frequency.

The dataset created in this thesis contains only nominal features, meaning that there is no logical ordering between its values. For example, google.com and facebook.com are both possible values for the Referer feature, but none is conceptually larger than the other. With ordinal, or label, encoding, each category level is assigned an integer value. In the case of the cookie synchronization dataset, this would impose ordinal relationships in a situation where no such relationships naturally exist, thus very likely misleading the model.

One can apply one-hot encoding to solve this issue, a scheme that assumes no natural
encoding between category values. In the one-hot approach, the encoded feature is removed, and one new binary variable is added for each unique category level in that feature [29]. If, for example, there are 42 possible values for the Response feature, then one-hot encoding would ultimately substitute it with 42 binary-valued columns. A possible improvement over a standard one-hot encoding is using dummy one-hot encoding, which encodes N-valued categorical feature into N-1 binary features, thus removing the redundancy intrinsic to one-hot encoding. Specifically, if out of N binary columns created by one-hot N-1 are all zero, then the last binary variable must be set to one, thus making it simply redundant.

Although theoretically, one could apply one-hot encoding to the dataset created in this thesis, in practice, it is simply infeasible due to the so-called curse of dimensionality. In short, in high dimensional space, all samples are far from each other, which results in phenomena that do not occur in low dimensional space. When training on high dimensional data, many algorithms exceed their runtime or fail to converge, as showcased by Jackson et al. [28]. In their case, one-hot encoding of a dataset with 24 categorical variables resulted in 71,253 columned dataset, causing most algorithms to exceed their allocated run time.

Applying one-hot encoding to this project’s cookie synchronization dataset would expand the original 13 features into 10,651 features. Following a rule of thumb suggested by Theodoridis et al. [30], there should be at least five samples per feature to ensure that there are several samples with each combination of values, requiring a minimum of 53,255 training samples. Although with a 2:1 train to test size ratio, 85,480 samples would be available for training, which is more than needed, one could argue that the amount of data is significantly less than enough with that many features. Even if it was enough, with that many features and limited computational resources, the training algorithms are likely to exceed their runtime or crash, which was actually observed in the case of Logistic Regression and SVM models.

Due to the curse of dimensionality, one-hot dummy encoding is used only to encode columns that are of boolean type. In this specific case, dummy one-hot encoding is equivalent to label encoding and produces one binary-valued column per each feature encoded. The rest of the features are encoded either by frequency or mean target encoding.

Frequency encoding is a way to utilize the frequency of category values as their numerical labels. It means that each distinct value of a given feature is labelled with the number of times it has been observed in that particular feature. Frequency encoding is suitable in cases where the frequency is somewhat related to the target variable. One can argue that this applies to the cookie synchronization dataset. As seen in section 4.1, cookie synchronization requests are somewhat rare, and so the feature values which discriminate these requests from normal ones should also be rare. For example, Google’s doubleclick.com, a popular tracking domain, should have a higher frequency than cnn.com, as requests to DoubleClick are made on visits to many various websites. On the other hand, DoubleClick should have a lower frequency than, for example, Facebook’s Content Delivery Network because CDNs are meant to serve a large portion of web content. Hence, by assigning appropriate weights to frequency encoded features,
a model can distinguish between regular and cookie synchronization requests.  

Another encoding scheme utilized in this thesis is mean target encoding. In this encoding scheme, labels for categorical values are directly correlated with the target labels by replacing the feature values with the ratio of the posterior probability of the target given a particular categorical value and the prior probability of the target given all training data:

\[ E(f, c) = \frac{P(c \cap \text{target} = 1)}{P(\text{target} = 1)} \]

where \( f \) is the feature being encoded, \( c \) is a particular categorical value of that feature, and \( \text{target} \) is the sample’s label. Since the target is binary and can take one of two values (0 and 1) only, the prior probability of the target being true can be calculated by taking the average of target values in the training set:

\[ P(\text{target} = 1) = \frac{1}{N} \sum_{y \in Y} y \]

where \( N \) is the number of samples in the training set. Mean target encoding expresses the probability of the target label being true when a given feature value is observed.

The encodings are fitted on the train set and applied to both the train and test sets. As mentioned, all boolean features are dummy one-hot encoded. For each of the three investigated models, the rest of the features are then encoded with frequency and mean target encodings. Additionally, label encoding is tried with the decision tree classifier, as this model can work directly with categorical data, if it is in numerical form. The best performing model-encoding pair is chosen once the models are trained on different encodings.

### 4.3 Logistic Regression

The first implemented model is well known logistic regression. The dataset, loaded from a CSV file, is first split into train and test sets in a 2:1 ratio. The split is done in a stratified fashion so that the training and the test set have about the same percentage of each class as the complete dataset. A stratified split is put in place to ensure better coverage of both target classes in the test set and thus a more reliable estimation of model performance.

---

1In the example with doubleclick.com and cnn.com one can observe a need for non-linear models. After frequency encoding, labels for trackers are not expected to be in the tails of frequency distribution. In the example, a tracker is expected to have a higher frequency than a standard website but lower than domains such as Content Delivery Networks, making the samples not linearly separable. Hence frequency encoding is expected to work better with Decision Trees, which are non-linear classifiers, rather than with Logistic Regression or Support Vector Machines (unless a kernel trick is used).

2CSV file was generated directly from the PostgreSQL database, which held the generated dataset.
As stated before, the model is trained with both mean target and frequency encoding. Table 4.1 presents the performance of logistic regression models relative to the dataset encoding scheme. Moreover, Figure 4.2 presents the receiver operating characteristic curve for both encodings.

<table>
<thead>
<tr>
<th></th>
<th>Mean Target</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision Class 0</td>
<td>0.992</td>
<td>0.916</td>
</tr>
<tr>
<td>Precision Class 1</td>
<td>0.801</td>
<td>0.177</td>
</tr>
<tr>
<td>Recall Class 0</td>
<td>0.967</td>
<td>0.593</td>
</tr>
<tr>
<td>Recall Class 1</td>
<td>0.943</td>
<td>0.618</td>
</tr>
<tr>
<td>F1</td>
<td>0.979</td>
<td>0.720</td>
</tr>
<tr>
<td>Weighted Average F1</td>
<td>0.965</td>
<td>0.665</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of logistic regression classifier for mean target and frequency dataset encoding.

The fact that mean target encoding is better for a model based on logistic regression is self-evident. In the case of frequency encoding, many cookie synchronization events are classified as regular requests (false negatives), and many regular requests are classified as cookie synchronizations (false positives). This results in a low weighted F1 score of just 0.665, compared to 0.965 for mean encoding. The supremacy of mean target encoding is also confirmed by its ROC curve in Figure 4.2. Since ROC does not depend on the class distribution, it is a useful measure for evaluating classifiers predicting rare events, such as cookie synchronization. In this case, one can see that the curve corresponding to the mean target encoding has a larger area under it than the frequency encoded model (0.955 vs 0.605), again confirming the supremacy of this particular encoding.

4.4 SVM

Another explored model is the Support Vector Machine, or SVM for short. SVM’s objective is to find a hyperplane with the maximum margin, i.e. the maximum distance between samples of both classes, allowing for a more confident classification. What is more, SVM allows for applying the so-called kernel trick, thus mapping the samples into higher dimensional space and potentially making the originally inseparable dataset linearly separable. Such an approach should significantly improve model performance on frequency encoded dataset, which, as mentioned before, is expected to be inherently linearly inseparable.

The training set-up is similar to the one for logistic regression, except that now a grid search hyper-tuning is performed to find a model with optimal parameters. The dataset is again split with stratification in the 2:1 train to test ratio. Instead of holding out a separate validation set, which would reduce the available training data, 5-fold cross-validation is performed to evaluate the model on different hyperparameters.
Chapter 4. Training The Classifiers

Figure 4.2: ROC curves for logistic regression classifier with mean target and frequency encoding.

<table>
<thead>
<tr>
<th></th>
<th>Mean Target</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class 0</strong></td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>Precision</td>
<td>0.994</td>
<td>0.962</td>
</tr>
<tr>
<td>Recall</td>
<td>0.780</td>
<td>0.957</td>
</tr>
<tr>
<td>F1</td>
<td>0.961</td>
<td>0.971</td>
</tr>
<tr>
<td>Weighted F1</td>
<td>0.918</td>
<td>0.863</td>
</tr>
<tr>
<td>Macro F1</td>
<td>0.961</td>
<td>0.948</td>
</tr>
<tr>
<td>Accuracy</td>
<td>0.948</td>
<td>0.944</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of SVM classifier for mean target and frequency dataset encoding.

During the grid search, SVMs with different hyperparameters were trained and evaluated. Model parameters considered included different values of regularization parameter $C \in \{1, 0.1, 0.01\}$, kernel coefficient gamma $\gamma \in \{1, 0.1, 0.01\}$, polynomial degree $d \in \{3, 4, 5\}$ and of course different kernel functions - RBF, polynomial and linear.

For mean target encoding, the optimal model parameters were $C = 1, \gamma = 1$, with a cubic polynomial kernel, resulting in a model with a 0.963 weighted F1 score. For frequency encoding, the optimal parameters were $C = 1, \gamma = 1$ and a RBF function as the kernel trick. In that case, the weighted F1 score reported on the test set was 0.944. Although this might seem surprisingly close to the score for mean target encoding, one can conclude by looking at the rest of the metrics that frequency encoding is significantly worse than mean encoding and results in many false negatives and a relatively small number of true positives.

Although the weighted F1 score for mean target encoded data is slightly larger for a
basic logistic regression model (0.965 vs 0.962), one can see that the benefit of the SVM model is its higher True Positive rate, meaning that SVM is better at predicting cookie synchronization events. This is confirmed by the AUC score, which is larger for the SVM model (0.959 vs 0.955).

4.5 Decision Trees

One last model considered in this project is a decision tree classifier. In short, a decision tree is a flowchart-like tree structure where internal nodes represent decision rules, branches represent the decision undertaken, and leaf nodes represent the outcome, which is the predicted class.

Unlike logistic regression or SVM, decision trees can work with categorical data - it only has to be encoded in numerical form. Thus, label encoding is also considered as an encoding scheme for decision tree models. Otherwise, the training set-up is similar to both logistic regression and SVM - data is split in a 2:1 ratio to obtain train and test sets, and the classes are weighted to handle the issue of an imbalanced dataset. Similarly to SVM, the decision tree model’s performance can be altered by changing its hyperparameters. Thus, a grid search is again performed to find the optimal model parameters. These hyperparameters include:

- \textit{criterion} \in \{gini, entropy\}: the function to measure the quality of a split performed by decision nodes and select the criterion that partition the data in the best possible manner. Although, as observed by Raileanu et al. [31], the frequency of disagreement between the two functions is only 2\%, we still check if using one function over the other can yield better results.

- \textit{max\_depth} \in [10...50]: the maximum depth the decision tree is allowed to grow to. The classifier’s depth is a measure of model complexity - the deeper the tree
is, the more decision nodes it has. By increasing the number of splits, the model captures more information about the data, which might lead to overfitting. Thus, to improve the model’s ability to generalize, all possible \( \text{max depth} \) values in the range from 10 to 50 are considered. 50 was chosen as the upper limit, as this is the depth to which the tree grows with no constraints on its depth.

- \( \text{min samples split} \in [1...40] \): the minimum number of samples required to split a decision (internal) node. According to Mantovani et al., who performed an empirical study on hyperparameter tuning of decision trees, the optimal values of that parameter tend to be between 1 to 40 \[32\] \(^3\), which is the range adopted in this thesis.

- \( \text{min samples split} \in [1...20] \): the minimum number of samples required at a leaf node. This parameter is also used to control the model’s ability to generalize, ensuring that the tree cannot overfit the training dataset by creating many small branches containing just one sample. In their study, Mantovani et al. observed that the optimal values for this parameter tend to be between 1 to 20 for the \text{CART} algorithm, which again is the range adopted in this thesis.

By performing a grid search over all combinations of the above parameters, optimal hyperparameter values for different encoding schemes were discovered - their values are presented in Table 4.5.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Mean Target Encoding</th>
<th>Frequency Encoding</th>
<th>Label Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Depth</td>
<td>26</td>
<td>38</td>
<td>32</td>
</tr>
<tr>
<td>Min Samples Split</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Min Samples Leaf</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison of optimal hyperparameters for mean target, frequency and label encodings.

<table>
<thead>
<tr>
<th>Class 0</th>
<th>Class 1</th>
<th>Class 0</th>
<th>Class 1</th>
<th>Class 0</th>
<th>Class 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>0.989</td>
<td>0.922</td>
<td>0.983</td>
<td>0.922</td>
<td>0.981</td>
</tr>
<tr>
<td>Recall</td>
<td>0.989</td>
<td>0.922</td>
<td>0.983</td>
<td>0.922</td>
<td>0.987</td>
</tr>
<tr>
<td>F1</td>
<td>0.989</td>
<td>0.922</td>
<td>0.983</td>
<td>0.922</td>
<td>0.984</td>
</tr>
</tbody>
</table>

Table 4.4: Comparison of decision tree classifier performance for mean target, frequency and label encodings.

Table 4.4 shows the performance achieved by decision trees models after hyper tuning. Additionally, Figure 4.4 presents the ROC curves together with calculated AUC scores.

\(^3\)This observation applies to \text{CART} decision tree algorithm, which is the one implemented by \text{scikit-learn}, the library used in this project.
Chapter 4. Training The Classifiers

26

Figure 4.4: Mean target and frequency encodings for decision trees models.

Just as in the case of logistic regression and SVM, the decision tree trained on the mean target encoded dataset outperforms the rest. However, contrary to the previous models, the performances are much more comparable this time, with all the metrics being consistently close to each other. However, the mean target still seems to have the upper hand, with all metrics, including the AUC, being slightly higher. Specifically, mean target encoding results in more true positives and negatives and fewer false positives and negatives.

4.6 Choosing The Right Classifier

There are several metrics to be considered when choosing the final classifier. Of course, the most important ones are the performance metrics, including weighted and macro average F1 scores. The difference between weighted and macro average metrics is that the macro F1 score is the arithmetic mean of all the per-class F1 scores. The weighted F1 score is the mean of all per-class F1 scores weighted by each class’s frequency in the dataset.

Another critical metric is AUC - the area under the ROC curve. This metric is used so that the classifier’s performance can be compared to the previous studies in that area, particularly the one by Papadopoulos et al. [5], who used an AUC score to evaluate their model.

The last metric used is the training time. One of the goals of this study is to produce a content-agnostic model which a browser plugin can utilize to analyze and classify HTTP traffic on the fly. With this in mind, the time needed to train the model becomes a crucial user experience issue. Of course, one can argue that the model could be trained offline once and then deployed so that the end-user is not affected by the training time. However, the dataset contains categorical data, with each value having a unique en-
<table>
<thead>
<tr>
<th>Metric</th>
<th>Logistic Regression</th>
<th>SVM</th>
<th>Decision Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighted F1</td>
<td>0.965</td>
<td>0.963</td>
<td>0.981</td>
</tr>
<tr>
<td>Macro F1</td>
<td>0.923</td>
<td>0.918</td>
<td>0.955</td>
</tr>
<tr>
<td>AUC</td>
<td>0.955</td>
<td>0.959</td>
<td>0.955</td>
</tr>
<tr>
<td>Training Time (s)</td>
<td>0.56</td>
<td>49.04</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Table 4.5: Comparison of models’ performances on selected metrics. All models use mean target encoding.

coding. Therefore, if a request contains a feature value that has not been previously observed, for example, a new domain for the Referer feature, to classify that request, the dataset must be re-encoded and model re-trained. The longer the plugin is used, the rarer the need for such actions is, as eventually, more and more values of a particular feature have an assigned encoding. Nevertheless, initially, the plugin is likely to face those issues, in which case the training time becomes a significant factor from the usability perspective.

Although the largest AUC score is reported for the SVM classifier, its training time (49.04 seconds) is unacceptably high. The performance of two other classifiers is comparable, with the decision tree model having a slightly larger weighted F1 score and a slightly lower training time. Therefore, this model is picked as the final classifier used in the next stage of this project to create a browser plugin.

Interestingly, the process followed in this project resulted in choosing the same classifier type as in the case of Papadopoulos et al. - the only other study in the area of ML-based cookie synchronization detection. Their approach also utilized decision trees as the final classifier, reporting an AUC score of 0.946, slightly lower than the 0.956 achieved in this project. Of course, as their model is trained on an undisclosed dataset, it cannot directly be compared with the one created in this project. However, a large, similar AUC score can be used as a sanity check, verifying the correctness of design decisions described in this study.

4.7 Removing Redundant Features

So far, the modelling has been done with all 13 features described in Section 3.3.1. This section aims to identify those features in the created dataset that do not affect the model performance or affect it negatively.

One of the motivations for performing such feature selection is the principle of data minimization. As expressed by GDPR’s Article 5, the data controller should constrain the collection of personal information to what is relevant and necessary to achieve a specific task [33]. Although in this thesis, the dataset is created without the participation of human subjects and their data, the underlying goal of the developed ML model is to use it in a browser plugin. In that case, HTTP traffic data, which could be extremely

---

4 Following this argument, one could argue that the encoding time should also be considered when choosing the final classifier. However, mean target encoding proved to be superior to frequency encoding in all three cases and is, therefore, the only encoding scheme considered. Hence the encoding time is not included in the comparison, as it is the same for all three models.
sensitive, would be collected. Hence, to promote ethical data collection practices and adhere to the data minimization principle, in this section, the irrelevant features will be identified to be discarded in any future work.

Two statistical tests are performed to obtain the correlation between each of the features in the dataset and the target label. Since the dataset contains only categorical data, metrics such as Pearson Correlation Coefficient cannot be used. Instead, two other metrics are used:

- Pearson’s Chi-Squared Test, a test for independence between categorical variables. In particular, Chi-Squared Test compares the observed frequency of categorical data with its expected frequency. Small test values close to zero mean that the frequencies are similar, and thus the compared variables are independent, suggesting they could be removed from the dataset.

- Mutual Information, calculated between two categorical variables to measure the amount of information one can obtain from one random variable given another [34]. In the case of the cookie synchronization dataset, this metric calculates the reduction in uncertainty about the target label from learning the value of a particular feature.

Figure 4.5: Relevance of particular features as calculated with Pearson’s Chi-Squared Test (top) and Mutual Information (bottom).

Figure 4.5 presents the scores that each feature has been assigned, according to the metrics above. Although the top three most relevant features are the same regardless of the metric used, the overall results are different. Thus, to see which features are redundant, one must train the model with a subset of data and pick features that result in the best performance. To that end, the decision tree model is trained on subsets of the most relevant metrics. The size of those subsets is changed from 1 (meaning that a single, most relevant feature is used for prediction) to 13 (all features used). Figure 4.6 presents the results of this experiment.
Chapter 4. Training The Classifiers

Figure 4.6: Performance of Decision Tree model, as measured by Weighted F1 score and AUC, for subsets of features. The horizontal axis is the number of top-performing features selected. The vertical one is the model performance. The left subplot uses the Chi-Squared test to choose the most relevant metrics, and the right one uses Mutual Information.

The best performing subset of features was found using Pearson’s Chi-Squared Test. This subset, including all features but "referer_category" and "sus_headers", achieves an AUC score of 0.956, marginally more than the score reported for training with all available features (0.955). Although removing those two features does not significantly improve the model performance, they could and should be removed according to the data minimization principle.
Chapter 5

Proof-Of-Concept Plugin

In this chapter, the findings from the previous work are utilized to produce a usable proof-of-concept web browser extension. One of this project’s contributions, the Chrome plugin, performs a real-time, stateless (content-agnostic) cookie synchronization classification of the HTTP requests made during an individual’s web activity and presents its findings to the user. In this way, non-tech-savvy users could gain the means to learn about the third parties which actively share data about them.

5.1 Requirements

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software. [35]

The opening sentence of Ken Thompson’s famous Turing Award Lecture, Reflections on Trusting Trust, provides a good starting point to discuss potential ethical considerations of our plugin. In short, the moral of Thompson’s paper is that one cannot trust the code that they did not entirely create themselves.

However, using a plugin that analyzes HTTP traffic requires users to trust the developer’s intentions and the created extension. After all, this kind of software handles extremely sensitive data. Any security breach would result in severe issues for the plugin user, such as session theft or simply leaking personal data.

For this reason, a paramount requirement for the plugin created as part of this thesis is an ethically sound data storage policy. Existing browser extensions, such as Mozilla Lightbeam, store all user data, such as cookies extracted from the browser, on the client’s side. Moreover, they are often open-source, meaning that any malicious handling of user data, such as sending it to a remote server, would quickly be discovered by the community.

This study takes a different approach. In the previous chapter, a content-agnostic classifier was proposed and created. Hence, unlike Mozilla Lightbeam, the plugin created in this thesis does not need to store user data permanently to function and
classify requests. It only needs to look at the HTTP request once and classify it, after which the data can be discarded. Therefore, for ethical reasons, the browser extension created in this chapter adopts a strict no-storage policy, in which user HTTP data is deleted straight after requirement classification. Furthermore, suppose the proof-of-concept plugin happens to be developed in future work enough to be deployed. In that case, this requirement will also make the extension compliant with GDPR’s data minimization principle, which states that the data should be retained only for as long as necessary to fulfil the purpose for which it was collected.

Other requirements for the proof-of-concept plugin include:

- **Real-time operation:** the plugin should analyze HTTP requests made by the browser in real-time, meaning that as soon as the request terminates, i.e. HTTP response is received, that request should be classified, and the user privacy report should be updated.

- **User-friendly usage:** the plugin should be easy to operate for a user without any technical background. This requirement is of utter importance - all existing cookie synchronization detectors, such as OpenWPM or CONRAD, require a lengthy setup process involving proxies and scripting, which is well beyond the technical skills of the average Internet user.

- **User-friendly report:** the plugin should produce its privacy reports in a way that is clear to a user without a technical background. In particular, as the user browses through the web, the plugin should clearly show the development within the environment of parties exchanging user data to improve user awareness of that issue.

### 5.2 Design

The general flow of action within the plugin is presented in Figure 5.1. As one can see, its design follows a well know client-server architecture. In particular, the user-facing frontend runs in the Chrome browser, intercepting the requests made by the browser, and sending them to the server for classification. The reason for this separation of functionalities is twofold. First of all, classifying every request within the browser would introduce additional computational overhead on top of the user’s everyday web activity, thus introducing extra latency and possibly slowing down the overall web browsing user experience.

The second reason is machine learning classification. Browser plugins are supposed to be lightweight and thus have limited capabilities. One of them, which comes from the fact that plugins are executed within the browser context, is that only JavaScript code is allowed. Thus, a Decision Tree classifier implemented in Python must be run outside the browser, thus creating a need for a backend (server) environment.

As mentioned, the role of the plugin’s frontend is to capture the request/response pairs, send them to the server for classification, and present the user with a privacy report. All of that must be done asynchronously so that the classification does not interfere with the user's browsing activity. To that end, the plugin script listens to the outgoing
and incoming requests, packages them into JSON and sends them to the server, which exposes REST API endpoints. REST API has been chosen as it is lightweight and does not require the frontend to package the payload (in this case, requests to be classified) in any particular way, unlike SOAP, which requires the data to be delivered as XML. Moreover, REST API naturally reflects the architecture of the plugin, in which requests coming from the client are independently serviced by the server exposing endpoints with different URLs for servicing different types of requests.

Once the server receives JSON with HTTP request/response pairs, it extracts the features that the trained classifier considers when making cookie synchronization predictions. With the features extracted, an SQL database is queried for features encoding so that categorical data can be encoded into numerical values. At this point, a prediction can be made, the result of which is sent back, with a 200 OK HTTP response code, to the client.

Once the user-facing client receives the response containing a label predicted for a given HTTP request, it must be included in a privacy report. The only requirement for this report is that it should be easy to interpret by a user without any technical background. Since conducting a user study, which would tell what the optimal report looks like, is beyond the scope of this work, an approach that has been proved to be optimal in existing plugins is instead taken. The rendered report follows the approach taken by Mozilla Lightbeam [36]. It presents the privacy report in a graph, with nodes representing domains and edges representing an instance of cookie synchronization detected between two particular domains. As Mozilla Lightbeam is targeted at non-tech-savvy users, it is fair to assume that the visualization techniques taken in its case
5.3 Implementation

This section discusses the technical challenges, along with their solutions, encountered while creating a proof-of-concept plugin for cookie synchronization detection in Chrome. Of course, as this project considers only a POC version of such software, making it ready for deployment in the open world would require considering issues like handling multiple users, authentication or security. This project aims to show that a plugin raising privacy awareness around cookie synchronization can be created with the existing plugin framework integrated with Chrome. Possibilities of its further development are discussed in Chapter 6.

5.3.1 Client

Intercepting HTTP Requests  To send HTTP requests for classification to the server, they first must be intercepted in the browser. Normally this could be done by setting a proxy, acting as a man in the middle. Chrome extension framework, however, provides the developers with chrome.webRequest API to observe and analyze traffic and to intercept, block, or modify requests in-flight [37]. To use it, the ”webRequest” permission must first be declared in the plugin manifest JSON file, along with necessary host permissions, which specify which domains can have their requests intercepted. Hence, to make sure the plugin is capable of intercepting all HTTP traffic, the host permission is set to ”all_urls”.

Chrome’s webRequest API defines a set of events that follow the life cycle of an HTTP request. The extension uses these events to intercept web traffic and react with specific actions. In particular, three different events are considered in the client implementation:

- **onSendHeaders**, emitted when the request headers are about to be sent to the network.

- **onBeforeRedirect**, emitted when a redirect is about to be executed, indicating that this particular request is a redirection.

- **onCompleted**, emitted when a request has been processed successfully and the response headers, status and body are available.

Event onSendHeaders indicates that a request, with a specific requestId, is about to be made. By adding an event listener observing this event, the client can access all of that request’s information, including headers and body. In the client implementation, the event listener listening to onSendHeaders events includes a callback function, which packages the intercepted request into JSON format and sends it, along with the requestId, to the server, notifying it that a request has been started.

1Chrome automatically assigns every HTTP request a requestId which is unique within a browser session. Thus, these identifiers can be used to relate different events of the same request [37]
The `onBeforeRedirect` event is used to send redirection data to the server. Since one of the features considered by the previously developed classifier is whether a request is an HTTP request or not, that information must be obtained for each classified request. Listening to the `onBeforeRedirect` event allows just that - the callback function notifies the server that a request with a given id is a redirection, allowing it to take that information into account when predicting with a decision tree.

The final event that the plugin listens to is `onCompleted`, triggered when the browser receives a response for a particular request. The callback function included in the listener registered with `onCompleted` events intercepts the response headers, which the `onCompleted` event makes available, and sends them to the server, along with the id of the request to which the response belongs. This allows the backend to factor the response metadata into the classification process, as explained in section 3.3.1.

The request made to the server in response to `onCompleted` event is also the one that triggers the actual classification - the server response to that request contains the predicted HTTP label. It is so because all the information needed to classify a particular request becomes available only at this point. Therefore, as one can see, for each HTTP request made by the browser, three requests to the server are made - one to deliver the HTTP requests, one to deliver the redirection information and finally, one to deliver the HTTP response. Such implementation was implemented as three different events make the three distinct sets of information available. For example, requests headers and redirection data are not available anymore when `onCompleted` event is fired, and, of course, response headers are not yet available when `onSendHeaders` is emitted.

**Rendering Privacy Report**  
The server response to the request fired by `onCompleted` event’s listener carries a label with classifier prediction for a particular HTTP request. Once received by the plugin frontend, this label needs to be included in the privacy report, which is easy to interpret by a non-technical user. As mentioned, this thesis follows the approach taken by Mozilla Lightbeam [36] rendering the privacy report in a visual graph form.

A framework called `vis-network` framework is used to render a graphical privacy report. It is a lightweight visualization tool to display networks consisting of nodes and edges, working with all modern browsers for up to a few thousand nodes and edges [38].

One critical issue that needed to be resolved to include the classification outcome in a render visible to the user was passing the classification from an event listener to the extension homepage. Following the architecture of Chrome Extensions, event listeners registered with the three `chrome.webRequest` events run in so-called `service workers` - scripts that the browser runs in the background, in an environment that lives independent of any other window or tab [39]. Although this conveniently allows the extension to observe and take action in response to events asynchronously, it makes it impossible to render any graphics - service workers do not have a Document Object Model and cannot display HTML documents.

To resolve that issue, the plugin uses a so-called `message passing` between the `service worker` script, which intercepts the HTTP traffic, and the `content script`, responsible
for visualizing the privacy report as a network in popup.html. To that end Chrome’s runtime API is used. In particular runtime.sendMessage method is used to send a one-time JSON-serializable message from the service worker script to the content script [39]. Every time a positive label is received from the server (meaning that cookie synchronization has been detected), a message containing the id of the classified request and a pair of domains engaged in cookie synchronization is packaged in JSON and sent along as the message body.

Meanwhile, on the receiving end, an event listener is registered with runtime.onMessage event, allowing the content script to listen to the incoming messages. Every time a message is emitted by the service worker, the content script intercepts it, and the cookie synchronization data is extracted. The domains included in the message are then included in the network as nodes connected by edges, indicating that cookie synchronization between these two domains has been observed.

```javascript
chrome.runtime.sendMessage({ msg: "Classification data", data: JSON.parse(responseText) }, (response) => {
  if (response) {
    console.log(response)
  }
});
```

Listing 5.1: JavaScript code included in the service worker script to send a message to a content script.

```javascript
chrome.runtime.onMessage.addListener((request, sender, sendResponse) => {
  if (request) {
    // Add the data to the privacy report network graph
    if (request.msg == "Classification data") { ... }
  }
});
```

Listing 5.2: JavaScript code included in the content script to receive a message from a service worker script.

### 5.3.2 Server

The backend environment is used to perform a bulk of the plugin’s logic, including HTTP request classification. The server is created with Flask [40], a web framework for developing web applications. It was decided to use Flask as it is a microframework, meaning that its core functionalities are simple but extremely extensible, making it a good choice for a proof-of-concept project. As Flask is a Python framework, it is also straightforward to import and use a pre-trained decision tree classifier.

The server listens to POST requests from the browser or the client. It does it using port 5000 of the localhost, which is the hostname provided to the address of the local computer. Thus the address of the listening server is 127.0.0.1:5000. This setup allows the plugin to be deployed and tested locally, so that client and the server

---

2In the case of Chrome extensions popup.html is the name of default plugin homepage - when a user opens the plugin in a tab, the popup.html is the document displayed by the browser.
can communicate without actual deployment to the outside world. In particular, to communicate with the client, the server exposes three endpoints:

- `/request_started`: the endpoint to which the client sends an HTTP request to be classified, along with that request’s id.
- `/request_completed`: the endpoint to which the client sends HTTP responses, which are matched to previously initiated HTTP requests using a unique request id assigned by the Chrome browser.
- `/log_redirect`: the endpoint accepting redirection information.

The need for three separate endpoints is motivated by an HTTP request lifecycle in Chrome - each endpoint accepts different data needed for classification. When the client sends a request to the `/request_completed` endpoint, it means all data required to classify this particular request has been made available. At this point, classification is performed, and the server responds to the client with the predicted label. Once the classification is done, the server gets rid of all the data on that request, implementing the aforementioned strict no-storage policy.

One thing worth noting at this point is the order in which the requests made by the plugin client arrive on the server. This order is of course affected by the request lifecycle in Chrome - first the `onSendHeaders` event is emitted, only then `onBeforeRedirect` and `onCompleted` events can be fired, strictly in that order. Since the server runs on the same machine as the browser, it is guaranteed that the requests sent in response to those events will be delivered and serviced in order. It is so because there is virtually no latency associated with delivering the request from the browser to the virtual server running on the same machine.

However, in a deployed system the server would not run on the same machine as the user’s plugin. Instead, it would run on a remote machine servicing requests from many different browsers. In such scenario the latency cannot be ignored, as it can change between the requests due to other network traffic. Hence it would be possible that the requests coming from the client are not delivered to the server in the expected order. For example, `onSendHeaders` event would send the request to the `/request_started` endpoint before `onCompleted` event sends one to `/request_completed` endpoint, but due to external factors the latter request could arrive to the server first. The proof-of-concept implementation assumes that this is not the case. A real browser extension would have to solve that issue, for example, by waiting with responding to particular requests until all requests that were supposed to arrive earlier are serviced.
Chapter 6
Discussion

Some of the design decisions taken in this thesis are based on assumptions or have implicit limitations. Others, on the other hand, can be improved by future work. This chapter discusses these limitations and provides ideas for further development of this thesis.

Cookie Synchronization Dataset
Perhaps the most significant limitation of the created dataset is its size. The dataset created by Papadopoulos et al. [5], the only already existing study focusing on ML-based cookie synchronization prediction, trained their classifiers on a dataset containing 179 million labelled HTTP requests. As in this thesis, the dataset was created with limited resources, it contains merely 129,515 data points and should therefore be considered relatively small. One could argue that the insights developed from the assembled dataset are inferior to the ones developed by [5].

A possible solution to this problem involves a more realistic crawling behaviour of the bot collecting the data, and longer web crawls. Right now, the bot collects HTTP requests by sequentially visiting the top 2000 domains obtained from the Tranco list. Perhaps a more human-like approach would involve following links, randomly chosen from websites’ source code, rather than a pre-specified list. Moreover, the bot would occasionally pick another popular website from the Tranco list and navigate straight to it. If this solution was scaled to multiple bots, running in separate Chrome profiles, it would essentially emulate a multi-user study and thus increase the overall size of the dataset, as well as its general reliability. This approach was not implemented in this thesis because of the lack of computational resources, particularly RAM, required for running multiple Chrome instances on one machine.

Machine Learning Models
With the correct encoding strategies, the performance of machine learning models described in this thesis is comparable to previous work in that domain. The final decision tree model can predict 92% of all cookie synchronization events in the created dataset. Those requests that are false negatives, i.e. cookie synchronizations predicted as regular requests, can be correctly identified by augmenting the machine learning model with a heuristic cookie synchronization detector.
However, the underlying goal of using machine learning is to detect those cookie synchronization events that heuristic methods cannot identify. Those extra, previously undetectable requests would be found among false positives, which are the data points labelled as regular HTTP requests (negative class) but classified as cookie synchronizations (positive class) due to their characteristics. Out of 414 false positives predicted by the final decision tree model, 96 were made among trackers, such as doubleclick.com or amazon-adsystem.com. Since these requests also contained a suspicious string that looked like an encrypted user identifier, they were likely, in fact, encrypted cookie synchronizations.

The remaining 318 data points classified as false positives are unlikely to be actual synchronization events, as they belong to domains not connected with tracking, such as news outlets. These falsely predicted HTTP requests are noise, and any future work aiming at improving the machine learning models should focus on eliminating them.

A possible strategy for tackling this issue is to change the model from a binary classifier to one with three target classes. The first one is for requests with no suspicious strings, which thus obviously cannot be cookie syncs. The second one is for requests with suspicious strings that proved to be regular requests. Finally, the third one is for requests containing suspicious strings that proved to be cookie synchronizations. Such an approach would make it possible to better discriminate between requests that are cookie synchronizations and those that only look like them. In that scenario, the set of points labelled as the second class but predicted as the third is where the encrypted cookie sync requests would be found. Modification of the methodology described in previous chapters to include this improvement is trivial, as it only requires slight changes to the labelling procedure described in section 3.2 and changing logistic regression to softmax regression.

Proof-of-concept Plugin One of the critical future improvements of the created extension concerns its user interface. As designing it was not a goal of this thesis, it is currently limited to a minimum form that a non-technical user can interpret. However, as mentioned in Chapter 1, potential stakeholders of this project also include policymakers and privacy auditors. Hence, augmenting the existing form of privacy report with optional features targeted at more advanced users is paramount. Those can include various centrality metrics describing the created network of collaborating trackers, which, as described in the last year’s part of this project, can give meaningful insights into the structure of the network and the dominance of its specific members.

Another crucial improvement includes making the plugin memory persistent. Currently, if a browser is closed, the graphical report is discarded. It then has to be created from scratch, making it impossible to see how the environment of data exchanging trackers changes across multiple browser sessions. A possible fix is adding event listeners that observe when the user exits and opens the extension. When exiting, the object storing the displayed graph would be stored within Chrome’s internal, persistent memory, available to all plugins. When opened, the extension would load it from that memory, thus providing information persistence across multiple browser sessions.
Chapter 7

Conclusions

The overarching goal of this thesis was to design and implement an end-to-end methodology for creating a privacy-related tool detecting cookie synchronization. Specifically, the previous chapters describe a stateless, or content-agnostic, machine learning classifier packaged as a browser extension to identify privacy-intrusive HTTP requests within a desktop browser. To create this tool, three separate goals had to be achieved. First, a cookie synchronization dataset was created from data collected by a web crawler. The release of this dataset through university channels is one of the main contributions of this study, as no other cookie synchronization dataset has been made public before. The created dataset was then used to train logistic regression, SVM and decision tree models capable of identifying privacy-intrusive HTTP requests. Finally, the decision tree model was used in a functioning proof-of-concept Chrome extension, detecting cookie synchronization in real-time and presenting the user with a visualization of its privacy-related findings.

The first of the three goals above - creating a publishable cookie synchronization dataset suitable for training machine learning classifiers - was motivated by the current lack of such publicly available resources. Using Selenium to automate visits to 2000 most popular domains from the Tranco list, the implemented web crawler collected 129,515 HTTP requests and labelled them with a heuristic cookie synchronization algorithm, previously used by many other studies from the domain of web privacy. Although the features included in the created dataset are inspired by previous work in the cookie synchronization domain, this thesis describes and collects significantly more characteristics of HTTP requests, resulting in a strong performance of machine learning models trained on that data.

The created dataset was then successfully used to train three different machine learning models. By overcoming the issue of a heavily imbalanced dataset and considering different encodings of categorical variables, mainly frequency and mean target, these classifiers achieve performance comparable to or even slightly higher than the one reported by existing work, such as [5]. Out of all encoding schemes considered in this thesis, mean target encoding proved to be the best performing - classifiers trained on the dataset encoded in that way achieved consistently high weighted F1 and AUC scores up to 0.981 and 0.959, respectively. Although, in the end, frequency encoding proved to be
inferior to the mean target one, it was shown that with the correct classifiers, such as decision trees or SVM with a cubic polynomial kernel, this encoding could also result in good classifier performance and AUC of up to 0.932.

Although the best performing model, according to the AUC score, was a Support Vector Machine, due to practical reasons, such as significantly shorter training time, a decision tree classifier was used as the final model to detect cookie synchronization. To achieve the final goal of this thesis, this model was ultimately used within a functioning proof-of-concept Chrome plugin. This browser extension was designed and implemented from scratch, based on a client-server architecture and REST API. Taking inspiration from existing plugins, particularly Mozilla Lightbeam, the cookie synchronization report was designed as a network of collaborating domains, updated in real-time as the user browses the web, making the plugin usable for individuals with no technical background.

In conclusion, this thesis contributes to the area of cookie synchronization and online privacy. The previous work in this domain is extremely limited - no cookie synchronization dataset has ever been made public. Moreover, the only study focusing on detecting cookie synchronization with machine learning ([5]) disclosed very little of their design decisions, such as how their dataset was encrypted or class imbalance handled. Finally, currently there exist no cookie synchronization detectors simple enough to be used by users with little to no technical knowledge. By releasing a cookie synchronization dataset, describing how machine learning models can utilize it and finally, developing a proof-of-concept browser extension, this thesis tackles those pressing issues. Hopefully, future work based on the contributions described in this report will advance the capabilities of open-source privacy projects and equip any person wishing to study their privacy at a more personal level with the right tools to do so.


[33] European Data Protection Supervisor, “Data minimization.”


