Tools for Active and Passive Network Side-Channel Detection for Web Applications

Michael Lescisin
University of Ontario Institute of Technology
michael.lescisin@uoit.net

Qusay H. Mahmoud
University of Ontario Institute of Technology
qusay.mahmoud@uoit.ca

Abstract

Since its creation, SSL/TLS has been the go-to solution for securing unencrypted web protocols - most commonly HTTP. The design of SSL/TLS, however, merely provides data stream encryption and authentication properties which often leads to the incorrect conclusion that by simply wrapping an unencrypted HTTP connection to a server with SSL/TLS, user privacy and web application behaviour integrity are guaranteed. Such type of information leak is unique in the sense that while certain web security vulnerabilities such as SQL injections have been well researched and thus there are known design patterns to avoid and penetration testing tools based on detecting known-to-be insecure design patterns, the state of research for the types of information leaks described in this paper still lags behind. In this paper, we discuss three design patterns that often result in side-channel information leaks along with three real-world websites which posses these vulnerabilities. Based on these three vulnerable design patterns we present a set of tools for detecting these types of side-channel information leaks given a training set of captured encrypted network traffic sessions.

1 Introduction

Secure Sockets Layer (SSL) and its successor Transport Layer Security (TLS) are cryptographic protocols which are concerned with encrypting a *connection* or *session*. The protocol begins with verifying the identity of one or more communicating parties, and if the identity can be successfully verified a *symmetric* encryption key is exchanged which remains private to both the client and server. This symmetric key is used to encrypt all traffic in the session between client and server as well as to generate *message authentication codes* (MACs) for ensuring the integrity of messages as they move through an insecure network.

Despite the use of encryption, observing an encrypted SSL/TLS session still reveals the following information: (1) approximate amount of data transferred in each session; (2) start and end times of each session; (3) IP address of client and server(s) as well as their domain name(s); and (4) order of sessions.

As will be demonstrated later on in the paper, this information can be, in many situations, sufficient for revealing the following *private* information about a user's interaction with an SSL/TLS protected website: (1) the web pages which the user has loaded; (2) private inputs which caused a web page to be loaded; (3) the relative and absolute time(s) which real-time event(s) occurred; and (4) contents of the web page based on how the page responds to window resizing.

A challenge when detecting side-channel information leaks in web applications is the difficulty to *generalize* the detection solution. As there exists a very wide variety of types of web applications, it is difficult to build a *general* model that could be applicable to all web applications. In many cases, human user behaviour would need to be accurately modelled. For example, estimating how long a user would spend on a web page before clicking a link, in order to build a probabilistic model of which page was loaded, is still a problem with no trivial solution. Furthermore, the requirements for *privacy* are not equal across all web applications as the threat model and information value often differs greatly across users and applications.

In order to cope with this challenge, instead of attempting to provide a *general* model of a web application, we instead focus on three common design patterns in web applications - specifically - feedback based on a user submitted form, real-time suggestion feedback based on *Asynchronous JavaScript And XML (AJAX)*, and *lazy-loading* of images. The novelty of our solution lies in the fact that, although we have not yet achieved fully automated and sufficiently accurate side-channel detection for web applications, we have created, evaluated, and released three models each covering a common

type of web application side-channel information leak, which we believe, will be of great help for security researchers to develop side-channel test cases for their web applications.

1.1 Threat Model

In order to keep the scope of this paper limited, we make the assumption that the adversary *only* has access to the *network communications* of a victim. Extracting information from the target application through more direct means such as SQL injection or compromised login credentials is not included in our threat model and is a topic for other research papers. Thus remaining within the threat model of our paper, we define *passive attacks* as attacks which require nothing beyond normal web application use by the victim and *active attacks* as passive attacks plus the ability to direct the victim to an attack site.

The threat of a *passive* attack is sensible to include in our threat model as this is the type of Man-In-The-Middle (MITM) attack which SSL/TLS strives to prevent. Practically speaking, MITM capturing of encrypted network traffic is an often attainable goal for many adversaries and may be conducted through means such as; Wi-Fi snooping at a public hotspot, compromising vulnerable internal network equipment to perform DNS/IP/ARP spoofing attacks, or obtaining data from malicious or compromised ISPs/VPNs.

Active attacks are also sensible to include in our threat model as getting a victim to load an attack site is often a feasible goal. This is especially true when the adversary is able to *modify* network traffic in an MITM scenario as many websites still use unencrypted *HTTP* connections. Thus is left open the scenario where an MITM attacked *HTTP* connection is exploited to cause information from encrypted *HTTPS* connections to be indirectly leaked to an adversary.

1.2 Our Contributions

We have implemented our solution using a Docker container of *Ubuntu 16.04 LTS* as the test environment, *Mozilla Firefox* as the testing web browser, and the Python module *scapy* for extracting features from network traffic captures. For evaluation, we have gathered three statistics from each classifier - true positive rate, false positive rate, and non-detection rate (when more than one outcome shares the highest probability). The merit of each classifier can be judged based on the maximization of its true positive rate and the minimization of both its false positive rate as well as its non-detection rate.

To summarize the research conducted in this paper, we present the following contributions;

- Show how *iSideWith.com*, despite being protected by SSL/TLS still allows an eavesdropper to learn which political candidate a user is most likely to vote for. To the best of our knowledge this website has never been tested before, academically, for side-channel information leaks. The vulnerability found with *iSideWith.com* is indicative of a much larger problem as it describes the side-channel associated with the implicit information flow which occurs when server response sizes are dependant upon user submitted data. The *iSideWith.com* example highlights the fact that an online recommender system designed without careful attention to side-channel prevention is likely to be vulnerable to side-channel leaks.

- Demonstrate that the auto-complete functionality within Google is still vulnerable to network traffic side-channel information leaks. The example of exploiting this side-channel against *Google Search*, while easily relatable due to the popularity of this search engine, is also indicative of a much larger problem - specifically the problem of side-channels in real-time user-interactive web services. As these types of services generate network traffic almost immediately in response to user events, information can be learned about the traffic causing user event. We believe that many types of real-time user-interactive web services such as online games or remote desktop connections (eg. noVNC) would also be vulnerable to this type of side-channel information leak.

- Extend the research on the auto-suggest side-channel to demonstrate how search term censorship could be implemented.

- Extend the current state of network traffic based side-channels by considering the effect of *lazy-loading* combined with pop-up window resizing.

- Demonstrate how exploiting *lazy-loading* could be used to estimate the number of items in a victim user's eBay shopping cart. The side-channel vulnerability found within the eBay shopping cart is indicative of a much larger problem - not only can an attacker learn whether or not a victim is storing cookies for a given domain, as the page loaded when cookies are present is often of larger size than the login page, they can also obtain information specific to a user. By loading a page in different sizes and observing the resultant changes to the generated

network traffic a unique pattern may emerge which can identify and track a user.

- Source code for side-channel detection and exploitation as well as relevant network traffic capture files have been released.

The rest of the paper is as follows; Section 2 will discuss the related work to our research, Section 3 will discuss the theory of the underlying design of our tools while Section 4 will discuss how the theory was applied in the creation of our tools. Section 5 will provide an overview of the supporting software packages that were used for the implementation of our tools. Section 6 will discuss results obtained by using our tools, Section 7 will discuss defences and countermeasures for improving user privacy on SSL/TLS protected websites, and finally Section 8 will conclude the paper and discuss future work.

2 Related Work

In this section of the paper we provide a review of the current state-of-the-art in research of network based side-channels in web applications. These papers have served as guides for developing our own side-channel detection methods and models.

In [6], the authors explain that due to the large variance in size of web resources, attackers are capable of learning a mapping between web resources and their associated patterns of generated encrypted network traffic. The authors then claim that, with the exception of anonymity networks, being able to identify web pages by analysis of encrypted traffic has not been a serious concern for most web developers. The rest of their paper serves to counter the claim that web page identifiability is not a serious concern for most web users. The authors discuss that as the web has moved from a collection of static pages *(Web 1.0)* to many dynamic web applications *(Web 2.0)*, a subset of the web application's internal information flows are exposed on the network. The authors conducted their research on a real online health website, a real online tax service, and a real online investment service.

In [15], a methodology (Sidebuster) based on static taint analysis is developed for detecting side-channel information leaks in web applications. *Sidebuster* works by a web application developer marking certain input information sources as *sensitive* for which *Sidebuster* then performs *information-flow analysis* to determine when the *sensitive* information is transmitted over an encrypted connection. When this occurs, the *Sidebuster* tool observes the encrypted network communication and determines whether or not the observable characteristics of the encrypted communication are sufficient to identify the *sensitive* data. *Sidebuster* is also capable of performing taint analysis for implicit information flows, that is, a *branch condition* satisfied by the value of *sensitive information*.

The same goal of research in [15] was followed in [5] but the approach is different. Instead of using a white-box static analysis methodology for determining when sensitive data will be sent over an encrypted connection, Chapman and Evans instead used a black-box approach thus avoiding the need for web application source code. Their method of side-channel detection consists of the web application developer defining the crawl through the web application and at each step in the web crawl, an AJAX supporting web crawler would record the associated Document Object Model (DOM) of the web page and use it as a state identifier. This state identifier is then used for building a finite state machine where each state corresponds to a web document DOM state and each transition corresponds to the trace of network traffic which caused the state transition. The authors then built classifiers trained on the network traffic data associated with each state machine transition. Based on their recorded network data, the authors used the *Fisher criterion* to determine the *classifiability* of the data. The authors tested their methodology on *Bing Search*, *Google Search*, *Google Health* and the *United Kingdom's National Health Service Symptom Checker* and obtained accuracies ranging from 46.1% on *Google Search* to 96.3% for *Bing Search*.

While following the same direction of study of the above discussed related work, our work distinguishes itself by providing the appropriate matching between a class of side-channel vulnerability and the best suited machine learning classifier. For example in [6], the authors list *low entropy input*, *stateful communications*, and *significant traffic distinctions* as causes of side-channels but they do not, however, delve into the machine learning theory for detecting these *different* types of vulnerabilities. Our paper discusses *response dependant page loads*, *real-time feedback systems*, and *lazy-loading* as three web development patterns that can lead to side-channels. For each of these patterns we provide both the theory of why information is leaked in addition to a machine learning approach to detect this type of design flaw. Finally, unlike any of the related work, we have publicly released all source code and data samples.

3 Underlying Concepts

In this section of the paper, we provide an explanation for the underlying concepts that our research is based on. The concepts discussed in this section are primarily about computer networks and modern web development.

3.1 Network Traffic Features

As stated in the introduction section of this paper, in a standard TCP/IP network, when an HTTP request is made over SSL/TLS, an adversary is capable of observing, in plaintext, all layers of the network stack except for the *application* (HTTP) layer which is protected by SSL/TLS. Given that the vast majority of Internet connected Local Area Networks (LANs) operate either using wired Ethernet or Wi-Fi using Ethernet encapsulation, this paper will make the assumption that all network activity is bound by *RFC 894* which specifies that the maximum size of an IP datagram that can be sent over an Ethernet frame is *1500 bytes* [8]. As the minimal size of an IP header is 20 bytes [1] and the minimal size of a TCP header is 20 bytes [2] the absolute maximum size of a data payload that may pass through one Ethernet frame is therefore *1460 bytes*. This implies that if a web resource is requested that is of size *1460 bytes* or greater, at least one full length packet will be generated. Through our own experimental results we measured the largest data payload size to be sent in a TCP datagram to be *1370 bytes*. This is a significant feature that provides insightful information when searching for requested web resources within encrypted samples of traffic. In order to find the approximate sizes of web objects transferred over a stream of encrypted traffic we employ the algorithm found in Figure 1.

- Iterate through each TCP carrying network packet and measure its size
- If the size is not 1370 bytes
  - If the previous size was 1370 bytes
    - Add this size to the sequence byte counter
    - Yield the value of the sequence byte counter as an HTTP object size estimate
  - Set the sequence byte counter to this size
- Otherwise
  - Add 1370 bytes to the sequence byte counter

Figure 1: This algorithm estimates the sizes of HTTP objects downloaded in a session based on continuous sequences of 1370 byte TCP payloads.

By following this algorithm we obtain a set of features correlated with the approximate sizes of web resources transferred in each session of captured network traffic.

Although the above described algorithm is effective for identifying pages as can be seen in the subsection *Testing iSideWith.com* of this paper, there is another important characteristic of network traffic that should not be overlooked when performing side-channel detection - that is the *relative timings* between packets or sequences of packets within a stream of network traffic. A burst of network activity can reveal an abundance of information about the type of information which is being exchanged over an encrypted connection. Often times, a burst of network activity corresponds to user events being triggered in a real-time system. Measuring the time between bursts of network activity can provide insight into the sequence of user-triggered events. Measuring the amount of data exchanged in a burst of network traffic may also provide insight into what each real-time event is. As can be seen later on in this paper, in the subsection *Analyzing Google Auto-Suggest Traffic*, by using the amount of data sent from server to client in a burst of network traffic, as well as the total number of packets exchanged, as features, we were able to, with 74% accuracy, identify which of the top ten Google searches of 2017 a victim user had searched for.

3.2 Modern Web Development

Modern web development greatly differs from the classical *Web 1.0* development as modern websites now execute complicated programs of client-side code, in order to interact dynamically with both the user and the server. Unfortunately as these new features are added to modern Web standards, new security vulnerabilities are often introduced [14].

In this paper, we will focus solely on how modern web development standards can be abused for leaking information through network based side-channels. It is through the discussion, implementation, and evaluation of side-channel attacks against *modern* web applications that a large segment of our research contribution is made. Furthermore, we extend the current state of research by considering not only *passive* but also *active* side-channel attacks where an adversary is capable of directing a victim browser to URL endpoints of a targeted private website and observing encrypted communications thus obtaining side-channel information. This *active* attack is described in the subsection *Counting Shopping Cart Items*.

Modern websites have been designed to be responsive to the wide variety of network connections which they may be loaded over, as well as the wide variety of screen sizes which they may be displayed on. For example, consider the *Dynamic Adaptive Streaming over HTTP (DASH)* protocol [13]. If client-side JavaScripts executing in a browser tab controlled by an attacker are capable of slowing down the victim's CPU and thus causing a lower quality of video to be loaded in the active tab, the attacker will thus be able to create a covert communication channel by indirectly controlling the incoming video bitrate, and thus network traffic, to the victim's computer. The DASH protocol is an example of the overall emphasis which *Web 2.0* has placed on making web content available to the user as soon as possible. Another exam-

ple of this phenomenon is *lazy-loading*, where JavaScript is employed so that only resources that are displayed in the user's viewport will be downloaded. A consequence of this is that by altering the geometry of a user's viewport, an attacker, by observing the encrypted web traffic from the victim's computer, can obtain additional information about the content which the victim has loaded based on how it responds to changes in viewport geometry. In the *Evaluation and Results* section of this paper, we will demonstrate how *lazy-loading* employed by *eBay's* shopping cart can be exploited to gain information on the number of items which a victim user currently has in their cart.

The example provided by *lazy-loading* is just one example of a side-channel that is a result of the use of the popular *Asynchronous JavaScript And XML* programming paradigm. Due to the *asynchronous* nature of *AJAX* the occurrence of network traffic is generally well correlated in time with the occurrence of user events (eg. mouse click, window resize). In addition, if an *AJAX* HTTP request is used to update an element on a page and an adversary is capable of detecting and blocking this request, then the adversary may also be capable of blocking certain web application functionality, thus implementing a type of censorship.

4 Tools Design

This section of the paper discusses the functionality provided by our tools and how it applies the above discussed theory for the purpose of side-channel detection and exploitation. Examples of these design concepts in action can be found in the *Evaluation and Results* section of this paper.

4.1 Exploiting Response Dependant Page Loads

Consider the following two criteria; 1) the next web page to be loaded is dependant upon private information which a user has submitted to a webserver and 2) the set of web pages which are potential candidates for the next page to be loaded has a large variance in size. If both of these criteria are true then an adversary can, by simply counting the amount of encrypted traffic sent from server to client, learn which page was loaded. As this loaded page is dependant upon the private information submitted to the webserver, the adversary can also learn the private information which resulted in the loading of this page, thus violating the *confidentiality* which should be provided by the SSL/TLS layer.

Our tool is capable of exploiting this type of side-channel. To do so, we employ the web object size estimation algorithm described in the subsection *Network Traffic Features* and for each network traffic capture, we obtain a list of approximate web object sizes transferred from server to client in the encrypted session. Our tool then can use a machine learning classifier from *scikit-learn* [10] which provides the *fit()* and *predict()* methods for determining a model for learning and exploiting, respectively, the network traffic side-channel. A description of an application of this tool, including the evaluation of various classifier algorithms can be found in the subsection *Testing iSideWith.com*.

4.2 Exploiting Real-time Feedback Systems

This exploit tool is most effective for exploiting network side-channels where the timing of packets carries important private information. This most often refers to real-time *responsive* systems. For example, when using the auto-suggest functionality on a search engine such as *Google* every keystroke entry generates a burst of network activity which flows from client to server followed by a burst of network activity which flows from server to client and contains the suggested search items. Since every keystroke event generates a burst of traffic, by counting these bursts, an adversary can estimate the number of characters present in the search string as well as obtain a probabilistic estimate of what the character was based on the size of the returned suggestions list.

In order to achieve this exploit goal, our tool works by clustering sequences of packets into *network activity chunks*. A user may simply decide to count *network activity chunks* as a data feature and for certain applications this may be sufficient information for breaching user privacy. If this *chunk count* feature does not provide sufficient information, our tool can also extract the amount of data transferred in a *network activity chunk*. Many times, the information contained within the *chunk size* feature is sufficient to lead to a privacy breach. For an example of exploiting a real-time feedback system, please see the subsection *Analyzing Google Auto-Suggest Traffic*.

4.3 Exploiting Lazy Loading

This exploit tool takes advantage of information leaked through observing DNS queries for content delivery networks (CDNs) as well as the effect of window resizing. As different web pages have different amounts of data transferred from various CDNs, by measuring the amount of accesses and the amount of data transferred from each CDN an adversary can gain insight into which page was loaded. Our tool is capable of inspecting DNS replies to determine the IP addresses of CDN servers used in a session. The amount of data exchanged with

each CDN server in a session is also measured. By repeating this CDN data measuring test with differently sized windows we are able to obtain a profile which in certain cases is capable of providing insight into the content of the window. Our results discussed in the subsection *Counting Shopping Cart Items* show that by using our tool, an adversary can learn information about the content layout of a page.

5 Implementation

In this section of the paper, we discuss the underlying software components that were used for building our tools. The goal of this section is to familiarize the reader with the high-level design of our tools which are available on GitHub [11] as well as to provide guidance to researchers developing their own network security testing tools.

5.1 Docker Container

Docker was chosen as the lightweight container for keeping an instance of the Firefox web browser separate from the host system. This allows for automated tests to be executed on web applications as a background process and thus does not interfere with a users normal web browsing activity. In addition, we are able to observe and manipulate network traffic directly as it moves into and out of the container and therefore, once again, we are not interfering with normal network usage. Finally, using Docker greatly simplifies the process of container building and therefore we are able to distribute the files required for reproducing our tests [11].

5.2 Linux Kernel's Netfilter

Netfilter is a flexible network packet manipulation framework built into the Linux kernel [9]. It is implemented as a set of hooks that are part of the Linux kernel's networking code. Kernel modules may register with these hooks and can therefore be used to manipulate network traffic. One feature of the *Netfilter* system that is particularly useful for building network traffic analysis tools, such as the ones described in this paper, is the *Netfilter queue* module (NFQUEUE). This module allows packets to be redirected by *iptables* rules where they can be manipulated by userspace programs and then returned back to the kernel's networking subsystem. To simplify development of userspace packet manipulation utilities, there exists a Python module for interacting with NFQUEUE. This Python module is what is used in our tool for performing active side-channel attacks. As through NFQUEUE one may observe and manipulate network traffic, we were able to feed traffic from NFQUEUE into analyzers and then filter it accordingly. This is demonstrated in the subsection *Censoring Google Auto-Suggest* where traffic was observed for key search terms and the search was blocked based on the presence of these search terms.

5.3 Scapy

As our tool performs an extensive amount of work with analyzing and manipulating TCP/IP network packets, a robust library is required. This functionality is provided by the *Scapy* Python module which is instrumental in extracting information from network packets [3]. Using *Scapy* we are able to extract the length of the TCP payloads from a set of packets.

6 Evaluation and Results

To validate the design and implementation of our tools, we demonstrate how we were able to exploit side-channels to determine which political candidate a user is most likely to vote for; determine what a user is searching for on Google and optionally censor it; and estimate the number of items that are in a user's eBay shopping cart.

6.1 Testing iSideWith.com

iSideWith.com is an SSL/TLS protected website which recommends a political candidate to vote for based on how closely they are aligned with the user's views. The user's views are obtained by answering a set of questions on divisive political issues. After the user submits their responses, *iSideWith.com* displays a page displaying the number one candidate followed by a ranking of how closely other candidates align with the user's views. A consequence of the fact that a large image of the number one candidate is loaded upon completion of the survey is that, due to the fact that the images of candidates vary significantly in size, the observed network traffic pattern, despite being encrypted, *will* be distinguishable for the various candidates. Thus, this web service leaks through a network side-channel, information about which political candidate a user is likely to vote for.

To demonstrate the exploitation of this side-channel we will consider the two most popular candidates for the 2016 US Election; Donald Trump, and Hillary Clinton. We have generated four sets of responses for each candidate which result in their recommendation from *iSideWith.com*. The result of running our test is 80 *PCAP* files with half corresponding to a recommendation of Donald Trump and half corresponding to a recommendation of Hillary Clinton. All *PCAP* files used for testing and

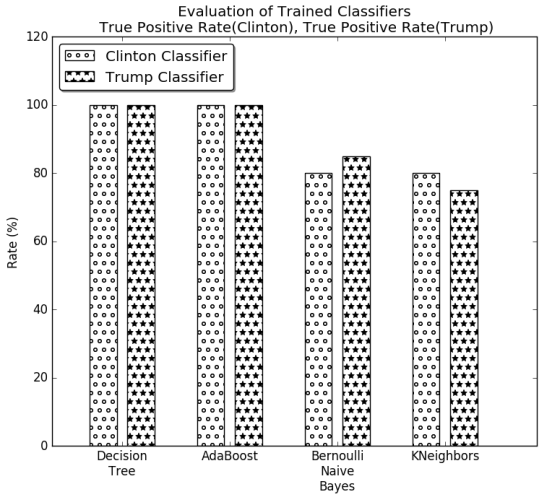

Figure 2: Testing the accuracy of various trained classifiers with network traces from isidewith.com.

training the classifier, as well as the source code for our classifier can be found on our GitHub page [11].

Evaluating the trained classifier against the set of test data demonstrates the fact that *iSideWith.com* is indeed vulnerable to a network based side-channel attack. The two best performing classifiers, *Decision Tree* and *AdaBoost* both gave perfect accuracies while *Bernoulli Naive Bayes* gave a true positive rate of 80% for detecting Clinton recommendations and a 85% true positive rate for detecting Trump recommendations. The *KNeighbors* classifier gave a 80% true positive rate for detecting Clinton recommendations and a 75% true positive rate for detecting Trump recommendations (Figure 2). When reading the figure it is implied that the false positive rate is equal to 100% minus the true positive rate and the non-detection rate is 0%.

In order to gain an understanding of how much traffic needs to be captured for training an accurate classifier we have examined the statistics of the existence of object size features within the captured training samples. Specifically, we build a set of all unique object sizes seen across all training samples. Then we check how many of these features are present in 100% of training samples, in at least 90% of training samples, in at least 80%, and so forth. Our results are as follows.

For network traffic samples corresponding to a recommendation of Hillary Clinton there were 378 unique features of which 7 are in 100% of samples, 11 in at least 90%, 16 in at least 80%, 18 in at least 70%, 20 in at least 60%, 25 in at least 50%, 30 in at least 40%, 40 in at least 30%, 71 in at least 20%, and 154 in at least 10%. For network traffic samples corresponding to a recommendation of Donald Trump there were 483 unique features of which 2 are in 100% of samples, 5 in at least 90%, 8 in at least 80%, 10 in at least 70%, 14 in at least 60%, 19 in at least 50%, 26 in at least 40%, 38 in at least 30%, 61 in at least 20%, and 175 in at least 10%.

Term	Mean	Std. Dev.	Min.	Max.
1	262.2	10.4	251	278
2	175.8	2.5	171	178
3	161.2	11.5	153	184
4	138.6	19.5	125	177
5	115.2	17.9	102	150
6	141.4	12.6	127	164
7	149.6	5.9	141	159
8	81.2	3.7	78	88
9	137.0	4.5	131	144
10	109.4	10.0	99	128

Table 1: Statistical parameters for the packet count feature from the Google Autosuggest training traffic samples. See Table 2 for the list of search terms.

6.2 Analyzing Google Auto-Suggest Traffic

In this evaluation, we demonstrate how the *auto-suggest* feature built into Google's main search page, is vulnerable to a network based side-channel information leak. To demonstrate this vulnerability, we have chosen the top ten trending Google searches of 2017 [7] and have trained various machine learning classifiers to evaluate their ability for correctly identifying a user's search based on the observed pattern of encrypted traffic.

Unlike the example where *iSideWith.com* was tested for side-channel information leaks, there is another important parameter that must be considered in this example, that is *time*. In our classifier, we have defined the maximum spacing between packets belonging to the same auto-suggest sequence to be *250 ms*. If this time limit is exceeded, then it is assumed that the victim has typed another key. For each *burst* of network activity, our classifier determines how many bytes have travelled from the server to the client and uses this as a feature. This feature provides insight into which character was typed. Due to the real-time nature of the auto-suggest functionality, network packets are seldom, if ever, carrying a full-sized payload. As each keypress event generates network packets, counting the number of packets exchanged between client and server is also a data feature that provides insight into what the victim user is searching.

As packet counting is a powerful feature for performing side-channel exploits on real-time feedback systems we have gathered statistical data from packet count features describing their *minimum value*, *maximum value*, *mean value*, and *standard deviation* from our captured network traffic used to train the Google Autosuggest classifiers (Table 1).

As can be seen from this data, certain search terms, especially those of similar length may have overlapping packet counts. Therefore simply training with one example of each query is insufficient and in addition, simply using the packet count feature will not give a certain predicted result. Using the packet count feature is a good estimator when the lengths of possible search queries has a large variance but additional features are required when lengths are of a more uniform distribution.

Considering exclusively the features associated with each *burst* of network activity, we performed Bayesian classification which provided a true positive rate of 38%, thus a small improvement over the 10% true positive rate of random guessing (Figure 3).

Considering exclusively the count of network packets as a feature and using a *Nearest Neighbour* classifier to determine the victim user's search provided a true positive rate of 62%, thus an improvement over Bayesian classification (Figure 4).

Our best accuracy was realized with using a hybrid approach of both *Bayesian* classification and *Nearest Neighbour* classification. By calculating the score for each element in the set of possible Google searches as its Bayesian classifier score divided by one plus the distance from the nearest neighbour classifier (adding one to eliminate a divide-by-zero error) a true positive rate of 74% was achieved (Figure 5).

Although the test performed in this section of the paper is limited to only ten Google search queries, it is important to also consider other information sources likely to be available to the adversary. Under our threat model where the adversary is capable of monitoring all network traffic, it is possible that the adversary builds a list of possible search terms based on either vocabulary collected from unencrypted *HTTP* connections or topics learned from side-channel cues such as those described in *section 6.1*. It is also important to note that the network traffic which follows the Google search is also likely to vary strongly based on what the search term was thus providing additional insight into the side-channel.

All *PCAP* files used for testing and training the classifier, as well as the source code for our classifier can be found on our GitHub page [11].

6.3 Censoring Google Auto-Suggest

As previously discussed in this paper, we make the research contribution of extending network side-channels from the exclusively passive domain where network traffic is only passively observed, to a hybrid domain where network traffic can be actively *manipulated* in addition to being passively observed. To demonstrate this, we will show how the *integrity* property of a web application can be violated through side-channel means.

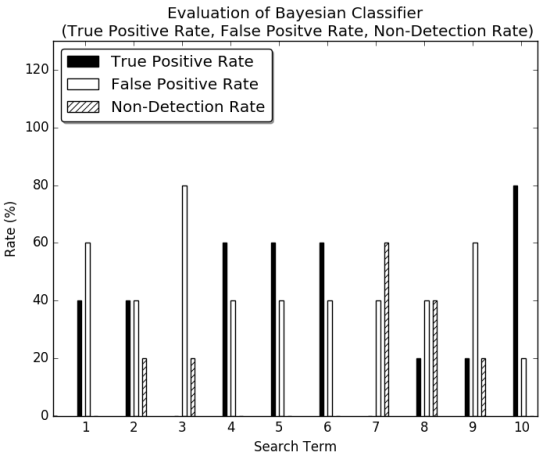

Figure 3: Testing the accuracy of the trained Bayesian classifier with Google auto-suggest network traces.

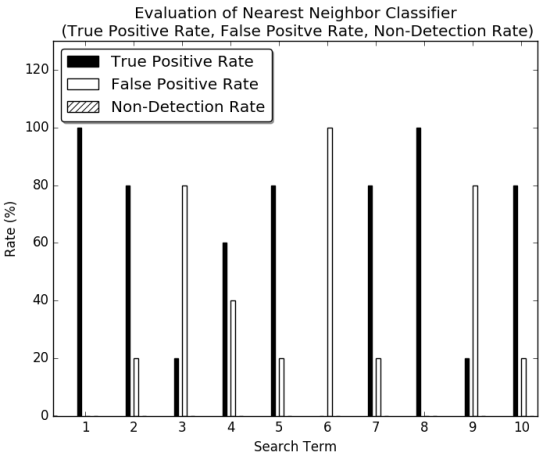

Figure 4: Testing the accuracy of the trained Nearest Neighbour classifier with Google auto-suggest network traces.

The main page for the *Google* search engine uses AJAX to provide automatic search suggestions for the text that is currently entered into the search bar. As has been previously demonstrated [5], an adversary is capable of observing the sequence of packet exchanges between a victim and *Google* and can use this information to determine what the victim is searching for despite the fact that *Google* uses SSL/TLS. In this evaluation, we expand this exploit to not only *observe* the behaviour of the victim but also to hide information from the victim based on an adversary's observations thus implementing *censorship* and *modification* of web application behaviour in spite of SSL/TLS.

To demonstrate the exploitation of this vulnerability we will describe how we have successfully accomplished the following task; when a victim visits the *Google* main page and searches for **hurricane irma** their connec-

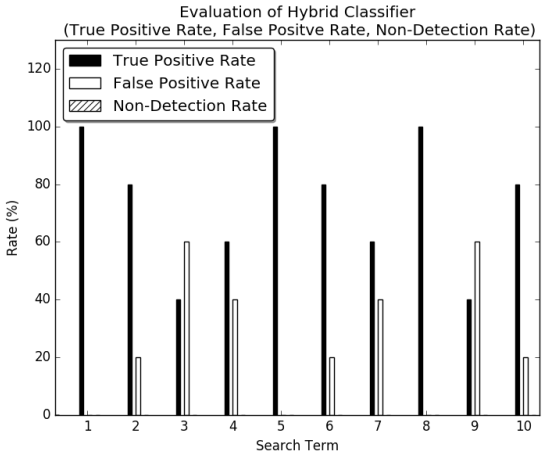

Figure 5: Testing the accuracy of the trained Hybrid classifier with Google auto-suggest network traces.

Number	Search Term
1	Mayweather vs McGregor Fight
2	Las Vegas shooting
3	Hurricane Harvey
4	Solar Eclipse
5	Matt Lauer
6	Fidget spinner
7	Aaron Hernandez
8	Tom Petty
9	Hurricane Irma
10	Super Bowl

Table 2: Search terms for Figures 3, 4, 5.

tion to Google is dropped while, using Google for other search queries does not result in this behaviour.

In order to implement this, we used the same detection algorithm described in the subsection *Analyzing Google Auto-Suggest Traffic* but instead of feeding it captured network packets from a *PCAP* file, packets were fed directly from the *Netfilter queue* (NFQUEUE). Once the condition of the hybrid classifier giving the censorable result after a range of packets have been exchanged, our censorship script would then call *nfqueue.NF_DROP* to block the victim's Internet connection.

The source code for our censorship script as well as the trained classifiers used in this example can be found on our GitHub page [11].

6.4 Counting Shopping Cart Items

This example demonstrates how *lazy-loading* can be exploited to reveal private information through network traffic side-channels. In this example, an attack website opens a pop-up window pointing to the URL of a

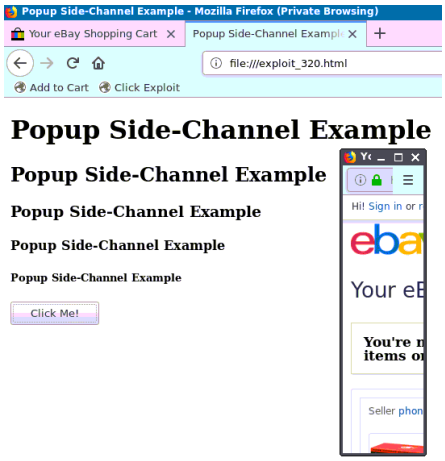

Figure 6: The exploit page opens by JavaScript a pop-up window of a given fixed size displaying the user's eBay shopping cart.

user's *eBay* shopping cart (Figure 6). Given that the attack website is able to control the window size of the opened cross-domain pop-up window, the attacker is therefore able to control how much of the pop-up window's contents is loaded via lazy-loading. *eBay's* shopping cart uses lazy-loading for loading product images only once they are within the page's viewport. Furthermore, these product images are loaded from the domain *i.ebayimg.com* and therefore by filtering traffic for that domain, we are able to obtain clearer information on which product images are being loaded. In addition, the browser cache must also be considered. If an image has been recently downloaded from *i.ebayimg.com* it will not need to be downloaded again. Combining the browser caching behaviour with the effect of pop-up window size on *lazy-loading* reveals the following rule - as the number of unique *lazy-loaded* images increases on a pop-up window, the number of requests to *i.ebayimg.com* also increases as the pop-up window is vertically enlarged.

In our demonstration, we conduct eight rounds of the test, the first round with one item in the eBay shopping cart, the second round with two, and so forth. During each test we open a pop-up window to the eBay shopping cart with the first pop-up window size being *280x280* and the last pop-up window size being *280x2840* with each test linearly increasing the height by 40 pixels. During analysis, for each of the eight rounds, the number of times *i.ebayimg.com* is accessed is counted. Our observations support our rule for browser caches and *lazy-loading*, in general as more items are added to the shopping cart, *i.ebayimg.com* is accessed more often as the pop-up window is vertically enlarged (Figure 7).

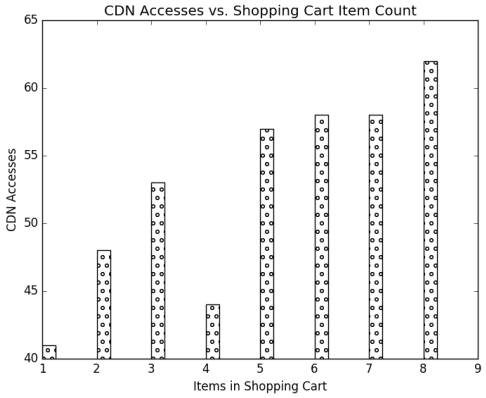

Figure 7: The number of accesses to the CDN generally increases as items are added to the shopping cart.

7 Defences and Countermeasures

In order to prevent both active and passive side-channel attacks on web applications and their users, two conditions must be satisfied; 1) network traffic patterns must be *indistinguishable* from one another and 2) the web application must either not change its behaviour as a result of network faults or the web application user should immediately be made aware of the traffic tampering.

Following the rule of making all network traffic patterns *indistinguishable* from one another could result in problematic performance overheads to the web application (eg. padding of smaller sized messages, generation of obfuscation traffic in real-time systems) and thus cannot be used as a general purpose fix to guarantee against the types of side-channel information leaks discussed in this paper. Instead, a *cost/benefit* approach needs to be taken. In [12], the authors propose a model for controlling which web resources should be enabled based on a *cost/benefit* analysis where cost is calculated as number of CVE reports, lines of code, and academic attacks related to this functionality and benefit is calculated as the number of websites which require the given feature for some user-visible benefit. For applying a similar *cost/benefit* model to the problem of side-channel mitigation, cost could be calculated as the value or sensitivity of the information leaked and benefit could be calculated as the gain in overall web application efficiency by choosing the side-channel possessing design pattern.

Another approach to side-channel mitigations in Web applications is to minimize the amount of *client-server* network trips necessary for proper web application performance. Although this could be done by carefully modifying the hand-written code of a Web application, a more elegant approach is to use model-driven development. This is compatible with the popular Model-View-Controller (MVC) Web application development paradigm which promotes component reuse. In [4], the authors discuss building web applications through a model-driven development process where their set of transformations generates *Angular.js* web application code from a model built from their UML profile. Although the paper makes no reference to side-channels, it is conceivable that elements could be added to their UML profile which provide side-channel resistant properties. For example, a web page object could be given a property such as *same-size* which would enforce that all pages of this property generate the same pattern of network traffic when loaded.

8 Conclusion and Future Work

In this paper we have introduced the theory and implementation for tools designed to detect side-channels in web applications. We have conducted evaluations which demonstrate how our tools are capable of exploiting web application side-channels similar to those discussed in older research papers and have confirmed that this research is still relevant to modern web applications. We have extended the current state of network traffic based side-channels by considering the effect of *lazy-loading* combined with pop-up window resizing and demonstrated how this could be used to estimate the number of items in a victim user's eBay shopping cart. Finally, we have released all our utility programs forming our set of tools and well as the relevant network traffic capture files on our GitHub page [11].

For future work, we would like to develop a model-driven web application framework similar to the one described in the section *Defences and Countermeasures* thus enforcing the rule that web resources served that are dependant upon private information must not provide any insight into the private information that resulted in their request.

References

[1] Internet Protocol. RFC 791, Sept. 1981.

[2] Transmission Control Protocol. RFC 793, Sept. 1981.

[3] BIONDI, P. Packet generation and network based attacks with scapy. In *CanSecWest 2005* (2005).

[4] CHANSUWATH, W., AND SENIVONGSE, T. A model-driven development of web applications using angularjs framework. In *15th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan, June 26-29, 2016* (2016), pp. 1–6.

[5] CHAPMAN, P., AND EVANS, D. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 263–274.

[6] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel leaks in web applications: A reality today, a challenge

tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 191–206.

[7] HARTMANS, A. Here are the top 10 searches on google in 2017, December 2017.

[8] HORNIG, C. A Standard for the Transmission of IP Datagrams over Ethernet Networks. RFC 894, Apr. 1984.

[9] JONES, A. Netfilter and iptables - a structural examination. In *SANS Institute InfoSec Reading Room* (2004).

[10] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[11] DNALAB TOOLS FOR NETWORK SIDE CHANNELS. https://github.com/uoitdnalab/networksidechannel, 2018.

[12] SNYDER, P., TAYLOR, C., AND KANICH, C. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. *CoRR abs/1708.08510* (2017).

[13] SODAGAR, I. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia 18*, 4 (Oct. 2011), 62–67.

[14] ZADEGAN, B., AND LESTER, R. Abusing bleeding edge web standards for appsec glory. In *Black Hat USA 2016* (2016).

[15] ZHANG, K., LI, Z., WANG, R., WANG, X., AND CHEN, S. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 595–606.