Decrypting encrypted web-search queries based on side-channel attacks

A report and practical illustration of the current vulnerabilities in applying encryption protocols for data transfer of web-search queries when exposed to side-channel attacks

Sebastian Blomberg

Alexander Jaballah

TDA602 Language-Based Security
Chalmers University of Technology
Gothenburg, Sweden 2016

Contents

1 Introduction

The great rise of Internet usage has enabled many new communication channels and other services that allows users to, for example, conveniently do their shopping, remotely communicate with their friends, or watch almost any movie from their connected devices. Many legit services give the impression that they are safe to use, and ensures that private information is protected during transmission; however, that might not be the case. Web applications contain sensitive information, for example health records, passwords, and other data that might be used to profile users. Research has shown that even legit website, using the latest encryption standards, may leak sensitive information through side channels during transmission [1] [2].

Standard web applications are divided into a client-side, i.e. the web browser, and a server side. To protect sensitive information within the web application during transmission between the two parts, the most common approach is to encrypt the network traffic. Even though this data is encrypted, sensitive information is still leaked, as will be explained in this report.

One common element in web applications nowadays is a search field that shows suggestions based on what the user has typed - a search field with so called *auto-complete* functionality. This functionality requires that the client-side communicate with the server-side to retrieve the list of suggestions to be displayed. Even though this communication is usually encrypted with HTTPS (Hypertext Transfer Protocol Secure), there are side-channels that leak information such as package size, IP addresses, ports and timestamps. This information does not seem like sensitive data, but after some elaboration it could just be enough to expose what the user entered into the search field, even though the data is encrypted.

Side-channel attacks is a well known phenomenon in the security world [3], and can sometimes be associated encrypted traffic, and the process to acquire valuable information from communication. There have been many different studies on side-channel attacks throughout the years, e.g. voice-over-Ip (VoIP), acoustic cryptanalysis, data remanance, and Row Hammer web traffic [2]. Even though side-channel attacks have been well known for many years, there are still many web applications that do not handle this issue. One reason for this could be that side-channel leaks have not caused grave enough damage to become well known in the web development field.

This report describes the work conducted to practically demonstrate and evaluate accuracy and efficiency of a common side-channel attack against search queries in autocomplete search fields. The goal of this project is to break HTTPS encryption and decrypt search queries made on one of the largest e-commerce site in the world, Amazon [4], as well to provide an idea of the accuracy and efficiency of the methods to do so. Furthermore, this report also describes and discusses mitigation and prevention techniques that can be applied by developers to reduce the risk of these types of side-channel attacks.

The rest of this report is structured as follows: Section 2 presents an overview of side-channel attacks and its impact on security, along with mechanisms to mitigate side-channel leaks. Section 3 specifies our goal in a more detailed manner. Section 4 provides an overview and a presentation of our methods to achieve the side-channel attack, as well as our implementation and its limitations. Section 5 provides the results obtained from our measurements to evaluate accuracy and efficiency. Section 6 discusses our results in terms of validity, and discusses the limitation of our approach. Section 7 concludes the report.

2

Background

This section provides background about side channels and side-channel attacks applied to encrypted web traffic, in order to give a theoretical understanding on the attack which is demonstrated in this report. This section also describes the elements, which are exploited, and mitigation and prevention mechanisms that could be applied to reduce the occurrences of web-based side-channel attacks.

2.1 Side-channel attacks in general

Side channels are, according to Zhou and Feng [5], defined to be the unintended output channels of a system. Side-channel attacks are thus attacks which utilizes unintended output channels, such as time, power consumption, electromagnetic signals, and sound, of a system (typically a cryptosystem). The main principle of side-channel attacks is the correlation between the internal state of a processing device of a system, and physical measurements taken during its operation [5]. For example, the amount of time taken to validate a password (physical measurement) might reveal something about how many characters of the password were valid (internal state), if the validation mechanism iterates over every character and terminates as soon as an invalid character is found. The measurement could then be used to reduce the possible amount of combinations in a brute-force attack.

The reasons side channels are present in cryptosystems is because the specification of a security protocol is oftentimes independent of the actual implementation [5]. This separation of concern enables theoretical mathematical design and analysis of security protocols without having to regard implementation details. Instead, certain assumptions are made about the implementation; for example that a cryptosystem is treated as a black box, meaning that its internal workings are not accessible to anyone, or that the medium over which messages are sent does not reveal any information additional. Although the benefits of this abstraction are clear from a theoretical perspective, there are certain drawbacks from a practical perspective. In practice, these assumptions are not always met. For example, encrypting web data may prevent unauthorized users from viewing its content, however it does not prevent unauthorized users from acknowledging the existence of data and transmission details of the data, such as timestamps, packet sizes, and sequence numbers [6]. This side channel could potentially be used to mount an attack.

2.2 Insecurities in encrypted data transmission over the Internet

As briefly explained in the previous section, one use of side-channel attacks is on encrypted web traffic. Much work has been carried out investigating side-channel attacks on Internet traffic. The essential part in most of that work is the exploitation of data encapsulation in the different layers in the Internet Protocol Suite (also known as the TCP/IP Model).

Popular Internet services, such as web browsers and mail clients, all operate using transfer protocols in the application layer of the TCP/IP Model (see figure 2.1), i.e. the application data is stored in the application layer. These services may dictate what transport-layer protocol to use (typically TCP or UDP), and also to a certain degree what Internet-layer protocol to use. However, the choices of these are limited; intermediaries, such as routers, must support the choice of protocols in order for data to be transferred. Most of these protocols transmit their headers in plain text. However, there are protocols, such as IPSec [7], which can provide end-to-end encryption over the Internet layer, thus protecting all application traffic. This is achieved by establishing links between pair of host/intermediaries and comes with some overhead, e.g. installing software. To our knowledge, such protocols are not widely used for public services, but rather used in corporate settings, for example when establishing a Virtual Private Network (VPN). Instead most popular services encrypt their data only in the application layer, thus leaving the lower-layer protocols' headers visible for any eavesdropper.

Most popular internet services have a secure version of their data transfer protocol. For example, the Hyper Text Transfer Protocol (HTTP), which is used when transferring web pages, has a secure version HTTPS, where the 'S' stands for secure. HTTPS uses Transport Layer Security (TSL), also commonly referred to as Secure Socket Layer (SSL) since TSL is a newer version of SSL, to encrypt the content at the application layer (including HTTP headers) [8]. A typical HTTPS packet is illustrated in Figure 2.2 below.

The size of the encrypted HTTPS data may vary depending on which encryption mode is used. If a stream cipher, such as RC4, is used the length of the encrypted data will be the same as the length of the plaintext; if a block cipher is used, the data will be padded so that the encrypted data size will be a multiple of the block size [2]. The choice of encryption mode is negotiated between the client and the server in a TLS handshake procedure [9]. No matter the choice of the encryption mode, the

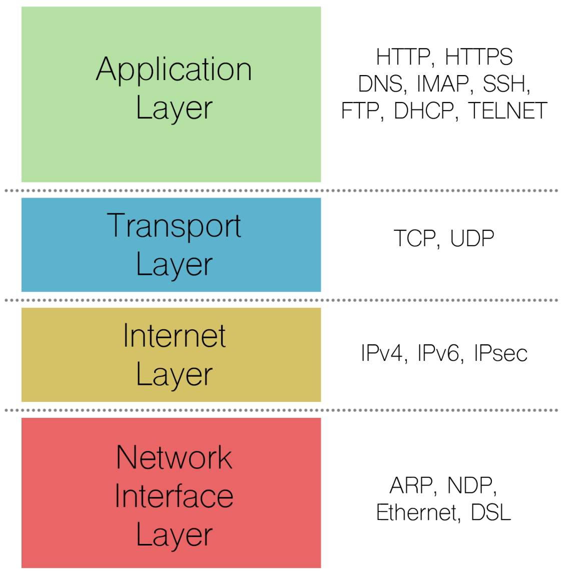

Figure 2.1: A graphical representation of the Internet Protocol Suite (TCP/IP Model)

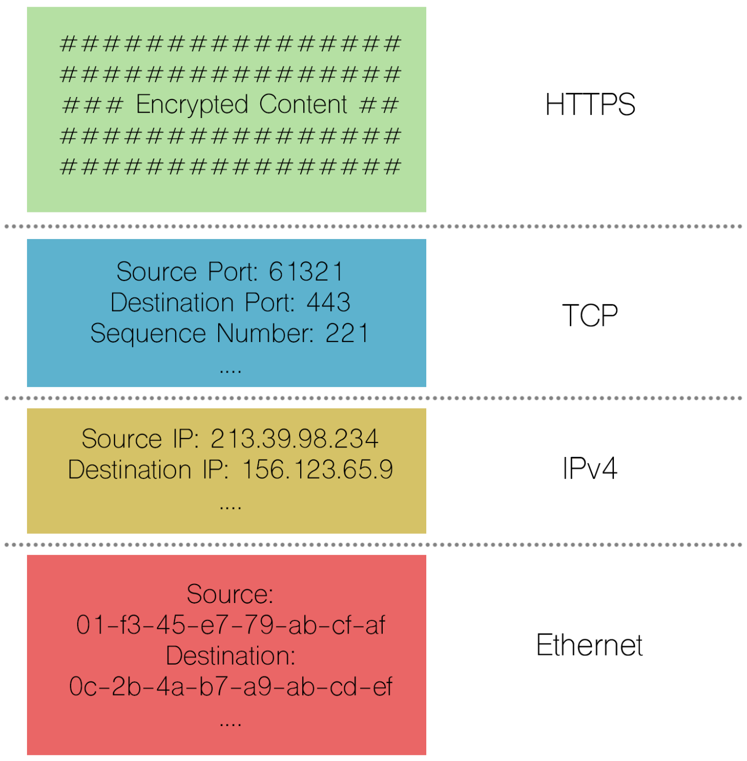

Figure 2.2: An example of an HTTPS packet. The content of the application layer (green) is encrypted. The content of the other layers can be viewed in plaintext

length of the encrypted data is still visible, along with source and destination ports, and the source and destination IP addresses in the lower layer packets. This can be considered as a side channel.

2.3 Exploitation of side channels in Internet traffic

Research has shown that quite trivial attacks, profiling attacks, can be mounted to retrieve the encrypted content of packets, with the use of the side channels described in the previous section. Profiling attacks generally consist of 3 major steps: collecting data, compiling data into a profile, and eavesdropping to obtain data that is compared to the profile. Depending on the purpose of the attack, the type of data to be collected will vary. In 2006, Liberatore and Levine [10] made a quite extensive profile of packet lengths associated with certain websites, as well as amount of visits to certain websites. During a month they tracked all DNS lookup requests for URLs in a university network and gathered the 2000 most accessed lookups. They then set up a web browser, connected to a proxy, which accessed these sites via a VPN, and logged the size of the packets while accessing the sites. Compiling this data into a statistical profile allowed them to predict which website was requested by eavesdropping on a VPN link. The predictions were made with $60 - 70\%$ accuracy depending on how old the profile was when the request was made.

Liberatore and Levine's [10] attack made use of a quite large population to gather data for their profile. However, if the HTTP traffic is not sent over a VPN, but instead just encrypted with TLS (HTTPS), recovering the content in the application layer is simpler. Chen et al. [2] describe deterministic profiling methods for successfully retrieving sensitive encrypted content from some top-of-the-line websites. In presenting their methods, they define a formal model for describing a web application. We adopt this model to be used when discussing and describing our work in this project.

Chen et al. [2] model a web application as a quintuple: $(S, \Sigma, \delta, F, V)$, where

- S is the set of states, both server-side (e.g. the state of a PHP program) and client-side (e.g. the Document Object Model (DOM) tree) that the web application can be in, excluding any back-end databases.

- Σ is the set of all valid inputs, such as keystrokes, mouse clicks, or information from databases.

- δ is the set of functions which transitions a web application from one state to another. These functions are defined as δ: $S \times \Sigma \rightarrow S$.

- F is the set of functions which model the web flow of a transition between states. A web flow has observable attributes, such as packet sizes, number of packets, direction etc. - all available in the lower-layer headers (e.g. TCP and IP headers). These functions are defined as $F : S \times \Sigma \rightarrow V$, where

- V is the set of web-flow vectors describing the observable attributes. For example, a flow vector $v = [\rightarrow 190, \leftarrow 1024, \leftarrow 578]$ denotes that one packet of size 190 was sent by a web browser to a server, and the server responded with two packets of sizes 1024 and 578 respectively.

According to this model, a web application can be considered as a state machine, with the states in S connected through the transitions in δ. Expressed in this terminology, the objective of an attacker looking to extract sensitive information is to find out which input σ_x the application received in a certain state S_x to make the transition $s_x \times \sigma_x \rightarrow s_y$ to another state s_y. To his help, the attacker has the flow vector v_x as a result from the transition $s_x \times \sigma_x \rightarrow v_x$. Below is a practical example of a simple web survey (for a graphical illustration, see figure 2.3).

$s_1 =$ *"An information page describing the survey. Contains an OK button to start the survey"*
$\sigma_1 =$ *"A click on the OK button"*
$\delta_1 = s_1 \times \sigma_1 \rightarrow s_2$
$s_1 \times \sigma_1 \rightarrow v_1 = [\rightarrow 50, \leftarrow 340]$

$s_2 =$ *"A question asking your gender with two options as buttons: male or female"*

$\sigma_{2,1} =$ *"A click on the male button"*
$\delta_{2,1} = s_2 \times \sigma_{2,1} \rightarrow s_3$
$s_2 \times \sigma_{2,1} \rightarrow v_{2,1} = [\rightarrow 60, \leftarrow 444]$
$s_3 =$*"Landing page for male participants, thanking them for taking the survey"*

$\sigma_{2,2} =$*A click on the female button"*
$\delta_{2,2} = s_2 \times \sigma_{2,2} \rightarrow s_4$
$s_2 \times \sigma_{2,2} \rightarrow v_{2,2} = [\rightarrow 60, \leftarrow 437]$
$s_4 =$ *"Landing page for female participants, thanking them for taking the survey"*

If an attacker observes a flow vector $[\rightarrow 60, \leftarrow 437]$, the attacker can make an inference that the participant is a female; hence, information about the participants gender could be retrieved even though the traffic was encrypted. This example assumes that the size of the flow vectors, and thereby the size of the data sent, is static and does not dynamically changed. Furthermore, it also assumes that the attacker can gain knowledge of which flow vectors are connected to which input and state, possibly by performing the survey himself and recording the flow vectors.

The example above is extremely trivial. It is seldom the case that the set of all flow vectors V contains distinct vectors. Consider for example if $v_1 = v_{2,2} = [\rightarrow 90, \leftarrow 543]$ in the case above. The ambiguity in the flow vector received by the attacker would lead to a 50% chance of determining whether the participant transitioned from s_1 to s_2, or from s_2 to s_4. However, if the attacker observes another flow vector of the same size, the sequence of those vectors would imply that the participant is now in state s_4. Thus, the attacker relies on reduction by sequences and distinction in flow vector sizes. Chen et al. [2] introduced a metric, density, for measuring the

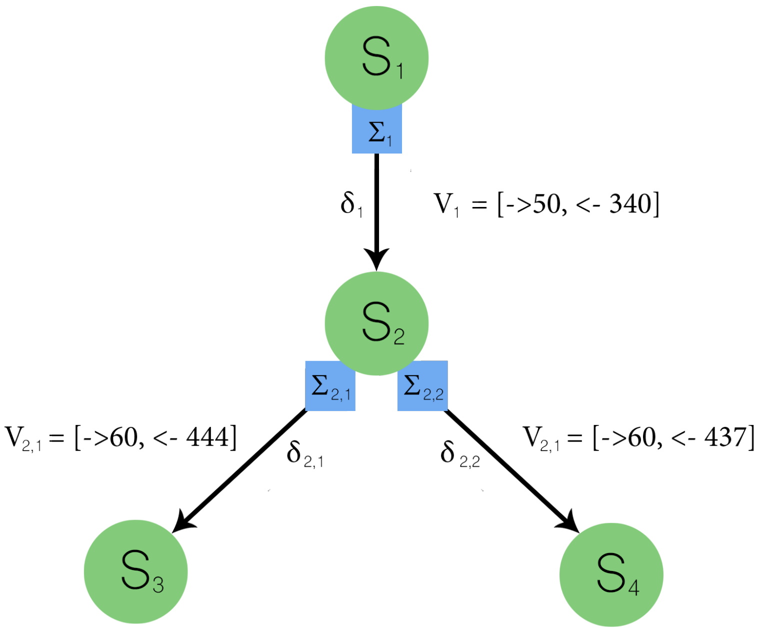

Figure 2.3: A graphical illustration, as a state machine, of the example described above

distinction of flow vector sizes, defined as:

$density(\phi) = |\phi|/[max(\phi) - min(\phi)]$

where ϕ is a set of packet sizes received. Generally, a density of less than 1 indicates that the packets are easily distinguishable [2].

2.4 Vulnerabilities in autocomplete functions

One modern-day web element which enables reduction by sequences, and possibly low density, is the suggestion box which pops up during free-text search [2]. This functionality is commonly referred to as autocomplete or autosuggest. An autocomplete function usually sends an AJAX (Asynchronous Javascript and XML) request to a server to retrieve possible suggestions based on what the user has typed. In traditional autocomplete functions, such a request is often sent after every keystroke that modifies the search query. As a consequence of this, the state of the web application transitions into a new state for every keystroke, even though the change in input data is very small (one character). This presents an opportunity for an adversary to enumerate all possible flow vectors and create a profile, since there is usually a quite limited set of possibilities. Through reduction by sequence, the adversary can then possibly retrieve the correct search query, see figure 2.4 below for a graphical illustration.

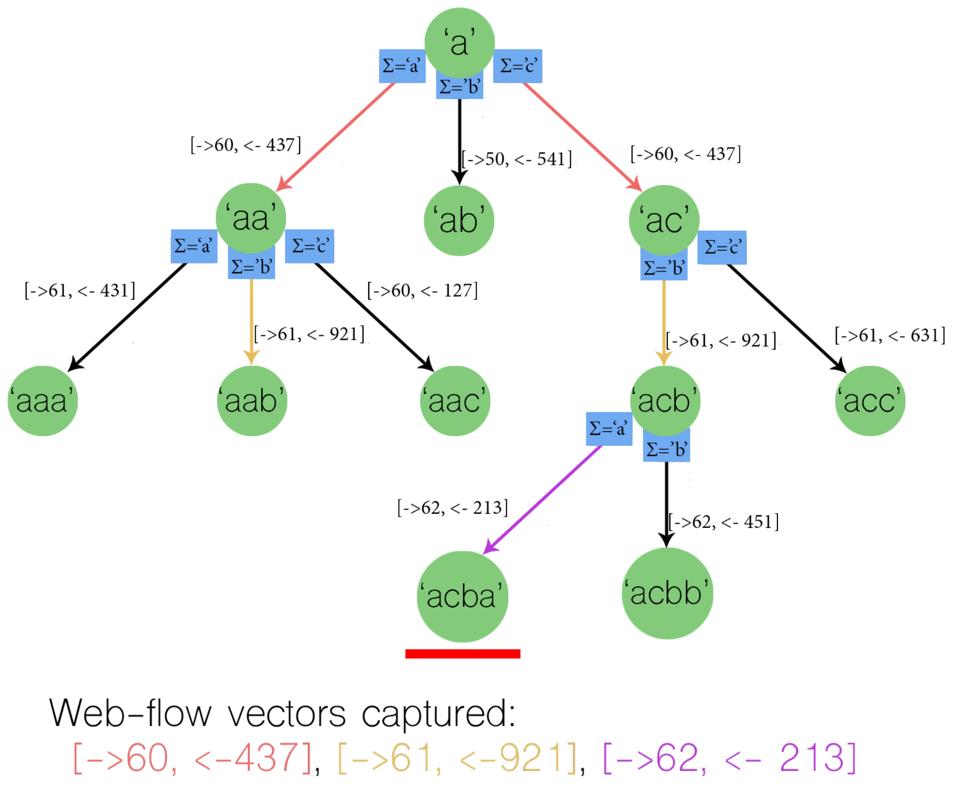

Figure 2.4: A graphical illustration of reduction by sequence. In the first state, the flow vector $[\rightarrow 60, \leftarrow 437]$ is received, and there are two possible matches ’aa’ and ’ac’. In the second state, the flow vector $[\rightarrow 61, \leftarrow 921]$ is received, and there are still two possible matches ’aab’ and ’acb’. In the third state, the flow vector $[\rightarrow 62, \leftarrow 213]$ is received and there is only one option ’acba’, thus the user’s input is decrypted with the help of reduction by sequence (assuming that the attacker has enumerated over all possible states to obtain a profile of the web-flow vectors all transitions).

Chen et al. [2] mounted a profile attack against an autocomplete function in a popular online-health application to discover medical conditions entered in a search field. They found a total of 2670 medical conditions and could easily identify the flow vectors. Furthermore, the density of the of the packets retrieved from typing ‘a’ to ‘z’ as the first character was determined to be 0.11, indicating high distinguishability. In fact, only two letters yielded the same flow vectors. The overall low density and reduction by sequence yielded favourable results for the adversary. This exemplifies the vulnerability of autocomplete when applied to a small and specific result set.

Chen et al. [2] also found that the search engine Google was vulnerable for this type of side-channel attack. In contrast to the online health app, which had only 2670 medical conditions in its result set, Google provides a significantly larger result set. Precomputing a profile for such a result set is a comprehensive task, the magnitude of which is polynomial; the amount of possible transitions for a state with input of maximum size n is defined by $x + x^2 + \cdots + x^n$, where x is the amount of characters to check, e.g. checking the characters 'a'-'z' would give $x = 26$. Although it is unlikely that all input would result in a transition to a valid state, the time complexity of pre-computing a profile is still $O(x + x^2 + \cdots + x^n)$.

Sharma and Bernad [9] suggested that a dictionary can be used to limit the amount of result that needs to be computed; However, they do not discuss how such a dictionary should be obtained, which could possibly be a complex question given the choice of language and also that many search strings do not consist of official words. There is however an alternative to precomputing a profile: a profile can be computed on-the-fly [2]. For example, as soon as a packet identified as an autocomplete packet is observed, all possible results for that state is computed, i.e. the server is queried for the web-flow vectors of the current input + 'a'-'z'. Inferences then be made about which character was typed. When the next packet is observed, all possible results based on the inferences about the all-possible previous characters are computed, and inferences about the new character can be made. In a best-case scenario, the amount of requests needed to compute the profile is:

length(search query ∗ size(set of possible characters)

The drawback of using on-the-fly profiles is the delay of having to compute profiles only after the packet is received. For attacks mounted against just a few users, this might work well. However, for attacks mounted at highly active network, the overhead of calculating on-the-fly profiles for every request received might be substantial.

No matter what type of profile used, there are some limitations for this way of deterministically recovering encrypted search queries. Firstly, every keystroke has to generate a single request, regardless on how fast the user types, i.e. if a user types to fast, instead of generating a single request for each stroke, some host-servers will generate a single request for the combined set of keystrokes. In such a case, it becomes difficult to distinguishing how many characters the user has typed. Secondly, and perhaps most importantly, the profiles may be dependent on user-specific factors, such as location, time of use, or whether the user is logged in. For example, search engines, such as Google, may alter the result set depending on location and language, but it may also tailor the results based on users' specific search history if they are logged in. As a result, deterministic profiling may only be possible under the conditions of these factors, and profiles may become obsolete quickly. Lastly, deterministic profiling may only yield result for queries where a non-empty result is returned. Every transition from a state s_x with an invalid input σ_x yields an empty-result state $s_{empty} \in S'$. If the flow vectors of these transitions are indistinguishable, which they most likely are, the input cannot be retrieved. For example: a user enters the character 'a' and retrieves a non-empty result set. The user then enters 'b' and retrieves an empty result set. It also happens that the query 'ac' will yield an empty result set. Thereby, there is no way of distinguishing whether the user entered 'ab' or 'ac', or any other query beginning with an 'a' that yields an empty result. See figure 2.5 for illustration.

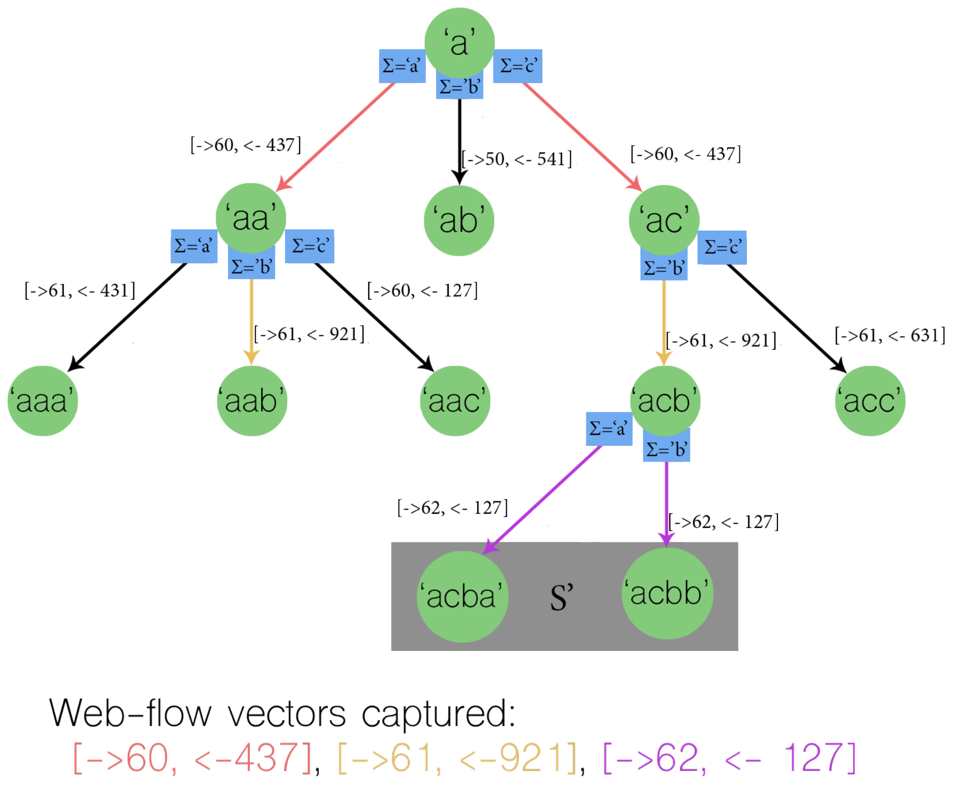

Figure 2.5: An illustration of the limitations of reduction by sequence. If the user enters a query which yields an empty result set, the web application enters any state in the set S', in which all states are indistinguishable by web-flow vectors.

Although the autocomplete functionality is relatively new, the issue of revealing information on every keystroke has existed for quite some time, and is not constrained to only autocomplete functionality. In 2001, Song et al. [11] showed that timing attacks could be used to retrieve encrypted data typed into an SSH session. SSH has an interactive mode, in which every keystroke, with a few exceptions, generates a packet sent to the server - similarly to autocomplete. However, SSH does not yield the same diversity in flow vectors, so the length of the packets is not enough to distinguish input. Instead, Song et al. [11] applied a statistical model (Hidden Markov Model) for determining the probability of pair of keystrokes, given the interleaving time between them. The result of this yielded, for example, a 50 times reduction of combinations to try when trying to crack a password by brute-force.

2.5 Prevention Mechanisms

Previous studies showed that one of the most common strategies is to implement packet padding. By randomly increasing the size of the message and retaining its direction, the observation of the packet is changed and cannot be deterministically predicted. Another way of padding that was mentioned in a study is to apply rounding, that rounds up the size of every packet size to 128-bit (i.e., padding the packet so that its size is a multiple of 16) byte [2]. The intention of this is to make every packet indistinguishable by size, independent of its content.

Schaub et al. [1] discovered a way to circumvent padding mechanisms used by Google. Google's packets included a token, a value that contained timestamp, and other random numbers. Furthermore, Google's packets were compressed using Gzip, thus the packet length became random. Even though this made it considerably harder, Schaub et al. still were able to retrieve information. This was done by using stack algorithms, and relying on a use of natural language, which drastically narrows down the set of possible answers. Therefore, only adding a randomizing packet length would mitigate the risk of side-channel leaks, but certainly not eliminating it.

Dummy and Split are two other countermeasures that could be used to mitigate side-channel leaks [6]. Dummy adds redundant files to the observation of a packet, while Split splits the file in the packet into smaller files. To achieve a more advanced countermeasure, one could use the HTTPOS method, which includes two parts of two previously explained countermeasures, Dummy and split. The first part, dummy, starts with injecting requests for objects and uses TCP maximum segmentation size, it also uses window options to decrease the packet sizes. The second part, split, consists of two different stages. The first stage, takes the starting x bytes of an object and splits them into equally sized packets. In second stage, the remaining n bytes are split into two different packets which has the sizes n and $n - r$, for a random $r < n$. The combination of both dummy and split results in an HTTPS encryption which will provide a higher level of protection.

A mitigation mechanism different from the above mentioned, could be to use Distributed Denial Of Service (DDOS) filters (DDOS filters). When the attack is run, multiple requests are sent to the host-server. Even though it is not the attack's main intention to overwhelm the host-server with a flood of request, the magnitude of polynomial requests could trigger a DDOS filter to block the requests and thus prevent our attack.

Side-channel attacks have been an issue for long time and to reduce every existing side-channel leak will demand a great effort. One could argue whether is worth the effort to try and reduce every possible attack. Instead Chen et al. argues that a more effective and efficient mitigation mechanism should be application-specific [2]. Thus, more responsibility lies with the developers, i.e. the developers has to identify the possible vulnerabilities for their specific web-application. Furthermore, it would require an analysis of the web-application's semantics, information flow and network traffic. However, to get an understanding of the attacker and the effectiveness of the defence, public domain knowledge is needed.

3

Goal

The purpose of this projects is to demonstrate a side-channel attack against a large e-commerce website, Amazon, to retrieve the plaintext of encrypted search queries using deterministic profiling of search suggestions that are presented to the user when typing search queries. Furthermore, we aim to evaluate the efficiency and accuracy of these attacks, specifically between precomputing profiles and computing profiles on-the-fly, to get an idea of what to expect from these kinds of attack.

4

Description of work

The work in this project consisted of launching side-channel attacks against the search field (see figure 4.1) on Amazon's British website, `https://www.amazon.co.uk`, in order to retrieve encrypted search queries. Although search history on Amazon might not be the most sensitive data (their US site does not enforce HTTPS), we wanted to demonstrate that even the world's largest e-commerce site [4] is susceptible to side-channel attacks that breaks HTTPS. Furthermore, it could be argued that search history on this site could be valuable information for competitors or for other parties trying to profile user's shopping behaviour or personality.

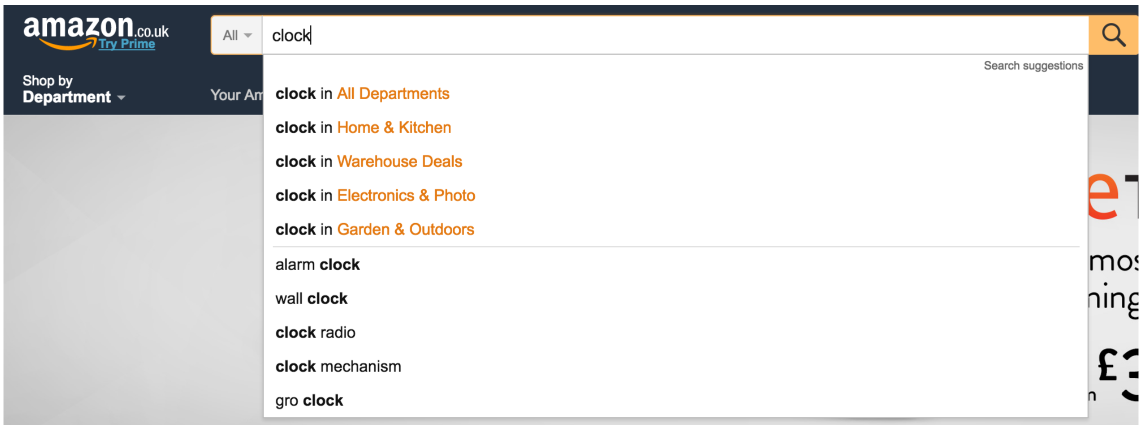

Figure 4.1: Amazon autocomplete searchbox

This section is divided into two subsections, a conceptual part, which describes our work from a theoretical perspective, and an implementation part, which describes the actual implementation used in launching the side-channel attacks.

4.1 Concept

The methodology of our attacks was derived from that of Chen et al. [2]. Essentially we assumed that we started at a state s_0, where the user has not yet entered anything into the search field. Given that the user enters any keystroke input σ_1 we aimed to find the state $s_1$to which the web application transitions and thus also decrypt the input σ_1. Similarly, for any coming sequential keystroke inputs σ_2, σ_3, $\ldots$, σ_n find the states s_2, s_3, $\ldots$ s_n. The retrieving of states was done through targeting a low entropy input element, the autocomplete field (which resulted in low packet density), as well as reduction by sequence. We considered the set of states S of the web application as every response to every possible combination of input to the search field while on the home page; all other states of the web application, such as viewing a product, or even searching while viewing products, were excluded.

We evaluated two methods for mounting attacks. The first method was to pre-compute (pre-generate) a profile of all possible states and the transitions between them - storing user input and flow vectors of every transition to every state. This profile was then used to lookup observed web-flow vectors (autocomplete packets) while eavesdropping on a network connection, and through the profile retrieves the current state and input. The second method was to compute a profile on-the-fly, as explained in 2.4, whenever an autocomplete packet was observed. Both of these methods were evaluated in terms of efficiency and accuracy. For both methods we also required that seven basic assumptions were met:

1. The IP address(es) of the host is known.

2. The destination and source ports, as well as the destination and source IP addresses of all packets are available.

3. High-frequency request spawning is accepted by the host.

4. Any user for which search queries are to be decrypted must be in the same region (in regards to IP addresses) as the computer mounting the attack.

5. Any user for which search queries are to be decrypted must not be logged in.

6. Every keystroke must generate a single request.

7. Every generated request must not contain any randomized element.

These seven basic assumptions also state the limitations of both these methods.

4.2 Implementation

The implementation was written largely in Java. To gain access to data packets on a network, the library pcap4j was used. pcap4j exposes an API (Application Programming Interface) to a native packet capture library, libpcap for UNIX-based systems, and WinPcap for Windows-based systems. Basically, the pcap4j library allows packets flowing over a network interface to be captured and viewed. The core component of the implementation uses this functionality to alert other parts of the implementation of incoming and outgoing packets. Listing 4.1 provides pseudocode for this core component, called the Sniffer.

Listing 4.1: Pseudocode for the Sniffer. The sniffer is a wrapper for pcap4j that filters based on IP addresses and port numbers

```
List listeners = <empty>
List ipAddresses = {178.236.6.47, 54.239.33.84,
   176.32.111.71}
int port = 443

pcap4j.setIpFilter(ipAddresses)
pcap4j.setTCPPortFilter(port)

while(pcap4j.waitForPacketToArrive())
     listeners.alert()
```

The sniffer is run in a separate thread to prevent disturbances that may cause packets to go unnoticed. The IP filter of the sniffer was set to only capture packets from one of the following addresses: 178.236.6.47, 54.239.33.84, 176.32.111.71 (all of which are amazon.co.uk's addresses), or the users' IP addresses. Furthermore, the port filter of the sniffer was set to only show packets carrying HTTPS traffic, i.e. port 443 [12].

The thread running the sniffer adds additional filtering to only allow packets generated as either an autocomplete request or an autocomplete response. Amazon uses Chunked Encoding [13] as encoding for HTTP messages. This implies that a message can be split up into several HTTP responses, with the line in every response's content indicating how long the response is. To indicate the end of a message, an additional HTTP response with content length set to 0 is sent. We used this as a token to indicate that an autocomplete message had been sent, since that response had a rather unique size of 37 bytes of application data (including HTTP headers). When such a response is received, the thread running the sniffer signals to anyone listening that an autocomplete packet(s) is available. This mechanism was then used in both the precomputing approach and on-the-fly computing approach.

4.2.1 Limitations

In order for the implementation to decrypt search queries, two assumptions have to be fulfilled in addition to those mention in Section 4.1. Firstly, we only interpret characters A-Z, since we are directing our exploit against an English website; to use more characters would increase the amount of possible permutations and thereby also computation time. Secondly, we assume that a user enters the search query sequentially at one once, and do not use the backspace key.

4.2.2 Precomputing approach

In the approach of precomputing a profile, two components were created: a state collector and an interceptor. The state collector is responsible for collecting and computing the profile (pseudocode of it can be found in Listing 4.2). The state collector iterates over all possible combinations of letters, which are specified as an input to the component. In our case we limited the letters to A-Z. On each iteration of a combination, the collector makes an asynchronous HTTPS request, using a WebRequestExecutor that can emulate a specified amount of web browsers. The HTTPS request is exactly like the one that would have been sent if the user had typed the combination in the search field. That HTTPS request has the following structure:

```
https://completion.amazon.co.uk/search/complete?
method=completion&mkt=3&client=amazon-search-ui&x=String
&search-alias=aps&q=<url encoded search query>
&qs=&cf=1
&noCacheIE=<current UNIX timestamp in milliseconds >
&fb=1&sc=1&
```

By simply changing the q parameter to whatever combination to be tried, the collector can emulate a user typing in the search field. The noCacheIE parameter exists so that Internet Explorer will not cache this AJAX request [14], however it is of no relevance to the actual result set returned, and therefore it does not need to be changed to the current timestamp.

Listing 4.2: Pseudocode for the State Collector

```
String inputAlternatives = "abcdefghijklmnopqrstuvwxyz"
WebRequestExecutor executor = new WebRequestExecutor(60
   threads)
State rootState = State("")
int maxDepth

collectWebFlowVectorsForChildren(rootState)
===========

collectWebFlowVectorsForChildren(State currentState, int
   currentDepth)
    for(Character c in inputAlternatives)
        executor.doRequest(currentState.name + c) {
            << on asynchronous response >>
            if isValid(response
                State child = new State(currentState.name
                    + c)
                if currentDepth < maxDepth
                    collectWebFlowVectorsForChildren(
                        child, currentDepth+1)
                currentState.addChild(child)
        }
```

The suggestion collector will search every possible combination until it reaches a pre-defined max length, or until it encounters an empty result. If it encounters an empty result, it will not search through any trailing characters of that combination. To determine if a packet contains an empty result set, it is evaluated against the characteristics of Amazon's empty result response:

completion = ["«url encoded search query»",[],[],[]];String();

If a packet contains an empty result set, its length will be equal to the length of the string above, evaluated with the current search query, plus 141 bytes of header + the string size of the length of the content (used for chunked encoding, as described above).

Finally, the result of the collection, i.e. the States, is saved as a JSON file for future use. We refer to this file as a profile.

The second component in the implementation for this approach is the suggestion Interceptor. Its implementation is quite trivial and illustrated by pseudo code in Listing 4.3. When run, the interceptor begins by loading a mapping, generated by the suggestion collector. It then uses the sniffer thread to listen for any autocomplete-response packets. When such a packet is received, the length of that packet (the web-flow vectors) is compared to the web-flow vectors saved in the profile of the correct depth level and produces all possible states (and inputs).

Listing 4.3: Pseudocode for the Interceptor

```
State rootOfProfile
while(snifferThread.waitForAutocmpletePacket())
    WebFlowVector v = snifferThread.getWebFlowVector
    List decryptedInput = findCombinationsBasedOnFlowVector(
       rootOfProfiles ,v)
    print decryptedInput
```

4.2.3 On-the-fly computing approach

The implementation for the on-the-fly approach is a combination the suggestion collector and the suggestion interceptor described in section 4.2.2. The on-the-fly component uses the same principles for detecting packets as the suggestion interceptor. When a web-flow vector is intercepted, the program queries the server for every possible combination of the current state, with regards to previous states, and matches the length of the web-flow vectors of the transitions against the received web-flow vector. For example, if no previous packet has been received, the program will query Amazon for the length of all web-flow vectors for characters A-Z and match those against the received vector. If a previous vector has been received and it matched two possible result sets, those for input ‘a’ and ‘g’, the program will query Amazon for all flow vector for the set sets: ‘a’+[A-Z], and ‘g’+[A-Z], and match those against the newly received vector. This process is illustrated in pseudo code in Listing 4.4.

Listing 4.4: Pseudocode for the On-the-fly Component

```
State rootOfProfile
List previousStates

previousStates.add(rootOfProfile)
while(snifferThread.waitForAutocmpletePacket())
    WebFlowVector v = snifferThread.getWebFlowVector
    findMatches(v)

void findMatches(WebFlowVector v)
    for(State s in previousStates)
        for(Character c in inputAlternatives)
            executor.doRequest(s.name + c) {
                << on asynchronous response >>
                if response.webFlowVector = v
                    previousStates.add(new State(s.name +
                        c, v))
                print s.name + c
```

4.3 Evaluating accuracy and efficiency

Part of the goal of this project is to evaluate efficiency and accuracy of this type of side-channel attack, and compare the on-the-fly approach with the precomputed-profile approach in terms of efficiency and accuracy. Several measurements were collected over a series of days in order to provide the necessary data to evaluate both methods. To evaluate the accuracy of this type of side channel attack, independent on which approach is used, 4 different metrics were used:

- Density d, as defined by Chen et al. [2]. Density was only calculated for the packets of the web-flow vectors in transitions from the start state s_0 to a state of 1 character, i.e. for any input 'a'-'z', and but not any coming input, e.g. 'aa' - 'az'.

- I_d - the amount of distinguishable input, such that $P((|S_n| = 1)|V_1, \ldots, V_n) = 1$, where S_n is the set of possible states after n transitions, given a sequence of web-flow vectors v_1, v_2, …, v_n; i.e. a state which can be uniquely identified given a sequence of web-flow vectors.

- I_v - the amount of valid input, $I_v = |S_v|$, where S_v is the set of all valid input that do not yield an empty result set.

- p - prediction rate, defined as $P(s_c|v_1, \ldots, v_n)$, where s_c is the correct state. I.e.

the probability of finding the correct valid state, given a sequence of web-flow vectors. The average prediction rate $\bar{p}$ is defined as the mean of the prediction rate for each state $s \in S_v$, with a maximum of n characters entered.

Measurements for the above metrics were calculated for every possible valid input consisting of 1 to 7 characters. The measurements were collected once a day for six days in a row, except for the measurements for 6 and 7 character inputs, which were only collected four times during six days.

In order to evaluate the specific efficiency of the two different profiling approaches, on-the-fly and precomputed profile, the average time to retrieve the nth ($n \in [1, 7]$) character of 10 randomly selected valid inputs was measured for the on-the-fly approach.

To evaluate the accuracy of the precomputed-profile approach, a degradation metric that specifies to which degree (percentage of original accuracy) a profile degrades over time was introduced. For example: at the time of a computation of a profile, a web-flow vector sequence $[[\rightarrow 62, \leftarrow 213], [\rightarrow 63, \leftarrow 412]$ would yield a set of possible states $s_1 = "aa", s_2 = "ab", s_3 = "ac"$. After a certain amount of time, the profile is re-computed and yields the following set of states for the same web-flow vector sequence: $s_1 = "aa", s_2 = "ab", s_4 = "ef", s_5 = "ei"$. The degradation rate would then be 50% since there is now only 50% probability that the old profile predicts the right state. In 33.33% of the cases, the old profile is going to predict s_3, which is no longer present. In 17.66% of the cases, the old profile is going to predict a false positive s_1 or s_2, while in reality the correct state is either s_4 or s_5.

Every measure was performed on a 2015 Macbook Pro 15" (16Gb RAM, 2,2Ghz Intel i7), connected to the Edouram wireless network at Chalmers University of Technology, Gothenburg. The measurements were collected roughly at the same time each day, however the exact times of measure varied up to 3 hours. Furthermore, the measurements were performed using 60 threads, each emulating a web browser and performing asynchronous requests as described in section 4.2.

5
Results

This section presents the results obtained from our measurements that evaluate accuracy and efficiency for both of the methods. This section is further divided into three subsections that describe the results obtained for accuracy, efficiency, and degradation of pre-computed profiles, respectively.

5.1 Accuracy of method

As described in section 4.3, the density was measured for the packets of web-flow vectors in transitions from the start state (an empty search field) to a state of 1 character. The result for a sample size of 38 was $d = 0,146754$, with a standard deviation of $\sigma_d = 0,00300$. The standard deviation is low, indicating that the lowest and highest packet lengths for the web-flow vectors remained roughly the same throughout the measuring period. Furthermore, as described in section 2.3, a density lower than 1 generally indicates that the packets are easily distinguishable. The obtained density therefore indicates that packets for transitions to a state of 1 character are easily distinguishable.

As specified in Section 2.4, the possible states for an input of size n is $26 + 26^2 + \ldots 26^n$. The obtained measurements in Table 5.1 shows that an input of size one and two yields 100% valid states (valid input), meaning that every input of size ≤ 2 gives a non-empty result. This rate reduces drastically as the size of the input increases. When having an input of size ≤ 7, only 0,009% of all possible input combinations yield states with non-empty result set, meaning that only 0,009% of all possible states can be inferred. Furthermore, we observed that the amount of distinguishable input, i.e. states that can be inferred with 100% certainty given an observed web-flow vector, is quite low for low-sized input. For example, on average only 43 out of 702 states can be inferred with 100% certainty for input of size ≤ 2. An interesting observation is that the ratio between I_v and I_d changes as the size of input grows larger, from being low $6,125\%/100\% = 0,06125$ for input size ≤ 2, to being high $0,005\%/0,009\% = 0,5555$. This is also reflected in the average prediction rate, as presented in Table 5.2.

Size of input	n	I_d	σ_{I_d}	I_v	σ_{I_v}
1	6	2,5 [9,615%]	0,54772	26 [100,0%]	0
≤ 2	6	43 [6,125%]	6,32456	702 [100,0%]	0
≤ 3	6	934,8 [5,114%]	24,99400	15160,4 [82,94%]	57,43083
≤ 4	6	15315,2 [3,223%]	442,60836	86426,33 [18,19%]	789,10625
≤ 5	6	94757,8 [0,767%]	775,91296	267605,75 [2,166%]	3429,91520
≤ 6	4	258084 [0,080%]	10659,62612	523902,33 [0,163%]	17669,33820
≤ 7	4	449419 [0,005%]	20672,65167	787133,33 [0,009%]	33857,15780

Table 5.1: Obtained measurements for all permutations of input of size 1 to size 7, where n is the sample size, I_d is the amount of distinguishable input, σ_{I_d} the standard sample deviation of I_d, I_v is the amount of valid input (input that results in a transition to a non-empty result set), and σ_{I_v} the standard sample deviation of I_v
. The value within bracket ([value]) is the percentage of the total amount of possible states for respective input size.

Size of input	n	$\bar{p}$	$\sigma_{\bar{p_n}}$
1	6	38,461%	0,0003%
≤ 2	6	7,1935%	0,8768%
≤ 3	6	6,2149%	0,1667%
≤ 4	6	17,731%	0,5810%
≤ 5	6	35,418%	0,6490%
≤ 6	4	49,255%	0,5446%
≤ 7	4	57,091%	0,2679%

Table 5.2: Obtained measurements for average prediction rate $\bar{p}$ for all permutations of input of size 1 to size 7, as well as standard sample deviation

As Table 5.2 shows, the prediction rate is initially quite high for predicting states with 1 character input, thus strengthening the indication by the low density. The prediction rate then drops for inputs of size ≤ 3. However, as the length of the input increases, the prediction rate also increases. For input ≤ 7, the average prediction rate has increased to 57,091% with a standard deviation of 0,2679%.

5.2 Efficiency of methods

The result of the calculation time to generate the different profiles, as described in Section 4.3 are shown in table 5.3. The results are also visualized in Figure 5.1.

Size of input	n	$\bar{t}$(sec)	$\sigma_{\bar{t}}$
1	6	0,01486	0,0023
≤ 2	6	0,94283	0,1616
≤ 3	6	21,8522	1,8482
≤ 4	6	520,067	142,60
≤ 5	6	1999,53	1265,9
≤ 6	4	9342,32	921,91
≤ 7	4	15755,6	478,97

Table 5.3: Obtained measurements for average computing time of a profile for inputs of size 1 to 7

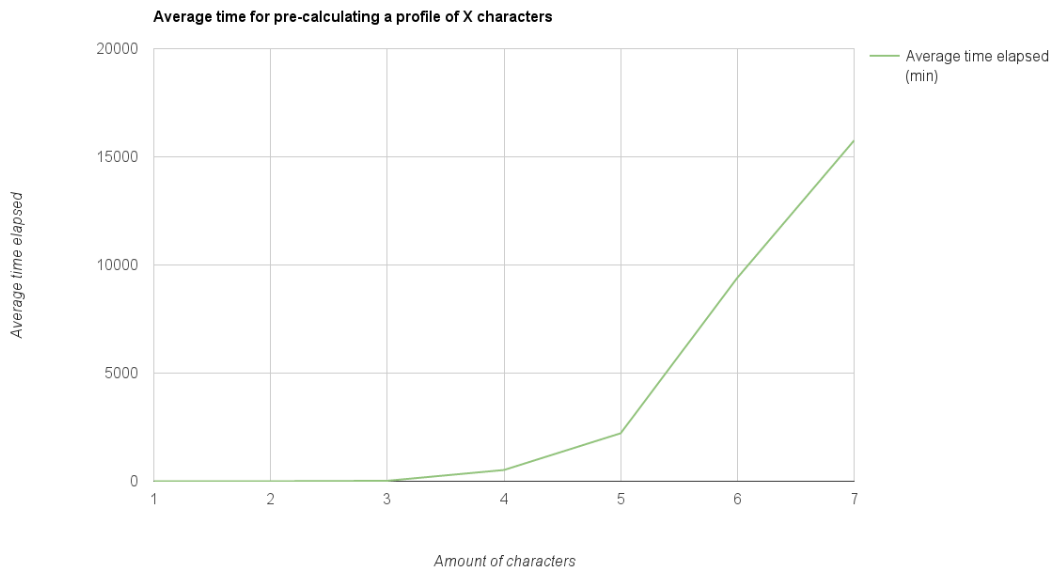

Figure 5.1: Average time for pre-calculating a profile of X characters using 60 threads

As a practical demonstration of scalability of the state collector component, results were also obtained for using 1 thread. These results are presented in Table 5.4. As a comparison with the measurements obtained using 60 threads, displayed in Table

5.3, we can see that using 60 threads instead of 1 does not exactly result in a 60x less generation time, but in proximity of 60x. For example, the generation time increase for inputs ≤ 2 is $52,19/0,94283 = 55,355$, the generation time increase for inputs ≤ 3 and $1278,82/21,8522 = 58,5213$.

Size of input	$\bar{t}$(sec)	$\sigma_{\bar{t}}$
1	1	1.71
≤ 2	1	52.19
≤ 3	1	1278,82
≤ 4	1	29203,46

Table 5.4: Obtained measurements for average computing time of a profile for inputs of size 1 to 7, using 1 thread

Measurements were also collected for the on-the-fly approach. These are the average times it took to compute an on-the-fly profile for the nth character of 10 randomly selected valid inputs. For example, the average time to calculate the profile to decrypt an 'e' that the user enters after having enterer 'appl', is 0,3025 seconds since 'e' is the 5th character in 'apple'. The average time to calculate any character n was calculated to be 0,2313 seconds. As can be seen in Table 5.5, the average time for all characters are quite similar. It is also noticeable that the standard deviation is relatively high in regards to the average time.

Amount of characters	n	$\bar{t}$(sec)	$\sigma_{\bar{t}}$
1	10	0,1044	0,0330
2	10	0,1886	0,06789
3	10	0,4264	0,1965
4	10	0,4458	0,2234
5	10	0,3025	0,1671
6	10	0,1517	0,03385
7	10	0,141	0,06681

Table 5.5: Obtained measurements for average computing time for 7 characters using the on-the-fly method with 60 threads

5.3 Degradation of pre-computed profiles

Pre-computed profiles are generated once, and can therefore become invalid if the web-flow vectors of transitions change on the website. To measure the validity of a profile we introduced a degradation metric d, as explained in Section 4.3. Figure 5.2 shows the validity $(1-d)$ for two profiles with input size 1 and ≤ 4, for 2016-05-23 11:26 and 4 days to come. We observed a 7-13% degradation of the $\leq$ 4 profile during the first 48 hours, and respectively 0-7% degradation of the 1 profile. After 48 hours though, a drastic drop in validity (high increase in degradation) occurred. The validity of the 1 profile dropped from 93% to 56%, and the validity of the $\leq$ 4 profile dropped from 87% to 10% - rendering the profiles effectively useless.

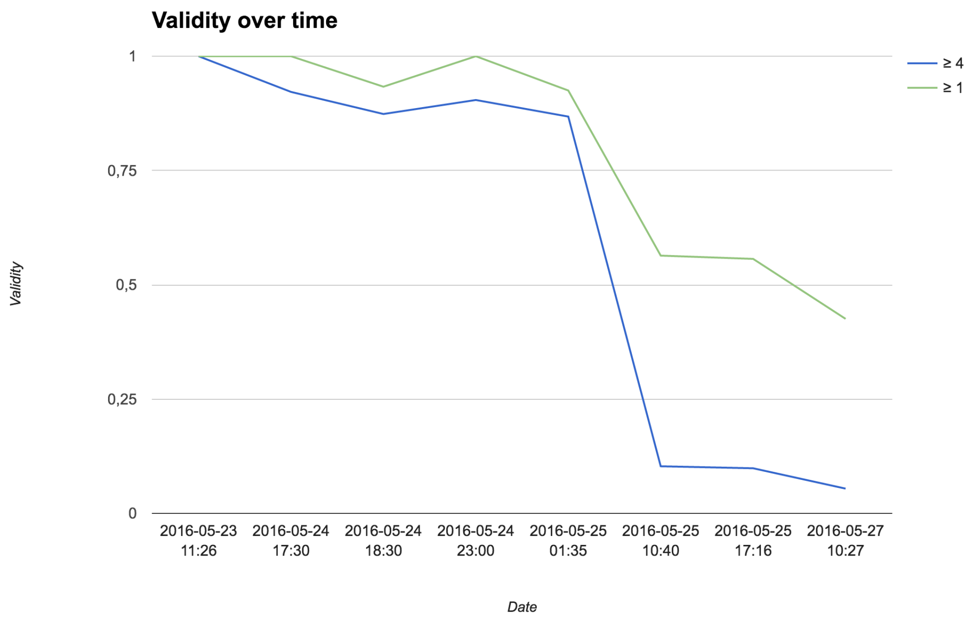

Figure 5.2: Validity of profile for input size of 1 (green) and validity of profile for input size f $\leq$ 4 (orange), in hours after 2016-05-23 11:26

6

Discussion

6.1 Accuracy

The results obtained for accuracy of this type of side-channel attack revealed that it is indeed possible to decrypt encrypted web-search queries made on the large-scale website Amazon.co.uk, with quite high accuracy. Whether that accuracy is sufficient enough depends on the area of application. For low-sized input, e.g. ≤ 4, the amount of valid input, I_v, is high, while the amount of distinguishable input, I_d, is low, making the prediction rate low. Being that profiles for low-sized input have a low prediction rate, it could be argued that they are not at all useful. This depends on the area of application; if it is enough to know a set of possible states, or if the exact state must be determined. Larger-sized input, on the other hand, have a higher prediction rate but they also suffer from having a lower I_v. In practice, this entails that fewer inputs can be decrypted, but the ones that can be decrypted will be so with higher accuracy. For example, the profile for input sizes ≤ 7 has a prediction rate of 57,091%, but only 0,009% of all inputs are valid - meaning that 99,009% of all input transitions into an empty result-set state and nothing can be inferred other than that the input is invalid. 0,009% is a very low number, however questions could be raised on how much value there is in decrypting invalid input. For example, if a user searches for "iphone" (valid input) there is value in obtaining that information, but it could be argued that there is no value in obtaining if a user searches for "ahsjus" (invalid input). Again, this depends on the area of application, but one should beware of the limitation that only valid inputs can be decrypted, exists.

There are methods, such as adding a dictionary or statistically modeling interception times briefly discussed in section 2.4, which could be applied to increase accuracy of this attack. We believe that even applying a simple dictionary of common searches would increase accuracy significantly. We noticed that when a user searches for common words, the set of possible input matches often contains one or two obvious choices, which often are the correct ones (see Listing 6.1 and Listing 6.2 for examples). By applying a simple dictionary, based on some sort of statistical inference method, the rest of the choices could be filtered out and the probability of finding the correct input would increase. However, this is beyond the scope of this project.

Listing 6.1: Example output from the on-the-fly interceptor when a user typed 'apple'. It is rather obvious that the most probable correct alternative amongst the set of suggestions is 'apple'

```
Intercepted packets on port 56055. Web flow vector: [<-629].
    Calculating possible states...
I think you typed 5 character(s) and I guess that you wrote
   one of the following queries:
appit
apple
applr
```

Listing 6.2: Example output from the on-the-fly interceptor when a user typed 'clock'. It is rather obvious that the most probable correct alternatives amongst the set of suggestions are 'clock' or 'cloth'

```
I think you typed 5 character(s) and I guess that you wrote
   one of the following queries:
elddi
elder
elnur
clito
clock
cloju
clote
cloth
nunat
nutbr
```

Our results were obtained over six days (except for the profiles ≤ 6 and ≤ 7 which were obtained over four days), making the sample size rather small. Furthermore, the standard deviations obtained for some metrics, for example $\sigma_{\bar{p}}$ for profile ≤ 2, were relatively high. Therefore, the validity of the accuracy results could be questioned. There is no statistical significance in the results and further work is required to gather this data. However, these results were intended to get an idea of accuracy, and not determine it exactly. In this aspect, we would argue that our results serve their purpose.

6.2 Efficiency

The efficiency results contained some interesting findings. Generating pre-computed profiles takes time and, as described in Section 2.4, in worst case scenarios it increases polynomially; however look-ups for requests can be performed in negligible time. In contrast, generating on-the-fly profiles gives a best case scenario of $length(searchquery * size(setofpossiblecharacters)$, however this needs to be performed for every search taking place. The question is then when to use what method. For example, the generation time for a pre-computed profile for input of size ≤ 7 took, on average, 15755,6 seconds (4,37h) to compute. The average time to compute an on-the-fly profile for any character in a random valid input was 0,2313 seconds. A simple calculation would yield that a pre-computed profile becomes beneficial only when attempting to decrypt $15755,6/0,2313 = 68117,6$ requests/second. However, this is naive way of looking at the problem.

In choosing between the use of pre-computed and on-the-fly profiles there are two main factors to account for: *bandwidth* and *degradation*. In the example above, we assumed that there was no limitations in bandwidth, while in reality there is. Our preliminary results showed that increasing the thread count to 60 from 1, thus emulating 60 web browsers making asynchronous requests, did not yield a 60x increase in generation time from. (These results should be cautiously regarded, since the sample size of the measurements with 1 thread only had a sample size of 1). This could give reason to suspect that generating a great amount of on-the-fly profiles simultaneously would result in a much longer generation time. Degradation of pre-computed profiles is also a problem to take into account. Our results show an example of how the validity of two profiles drastically dropped only after 48 hours. Pre-computed profiles seem to be sensitive to small changes in their web-flow-vector-to-state mappings, which is logical due to their tree structure. If the web-flow vector changes for a certain state, that state and all its children will be rendered useless. Therefore, if a change in the web-flow vectors from the empty start state s_start to any state $s_a - s_z$ occurred, it would affect all its children, rendering 1/26 of the profile useless. Our results showed an example of this, when a large degradation in a profile for input size 1 caused an even greater degradation in a profile for input sizes ≤ 4. The choice of profile then, is highly dependent on the area of application. If large amounts of search queries are do be decrypted on a highly active network, pre-computed profiles might be necessary. In most cases however, on-the-fly profiles should suffice and will be more beneficial since they do not degrade.

A different alternative to the two approaches presented in this report is a hybrid approach, i.e. building a pre-computed profile by generating on-the-fly profiles. The result of the generated on-the-fly profiles would be saved (cached) into an (incomplete) pre-computed profile. This will probably reduce the amount of requests that have to be made for generating on-the-fly profiles, while taking less time to generate than a complete pre-computed profile. The degradation problem would, however, still be present and have to be dealt with.

6.3 Practical Limitations of Implementation

Our implementation for performing this side-channel profiling attack has certain limitations that may not be practical. For example, every keystroke must generate a request. Amazon's autocomplete search field has a built in functionality that detects if two or more characters are entered within a certain amount of time (we estimate 200ms) and then only sends a request for the last character entered. Likewise, other autocomplete search fields on other sites may cache previous search queries to speed up response time, thus not sending a request for every character. As long as there is one request sent, our implementation could be modified to account for this behavior. However, it would require a lot more calculations to be performed, which could possibly take time and result in a larger set of matches. Another limitation of the implementation is that it does not take backspaces into account. This could potentially be realized. However, this would also increase the amount of calculations and increase the size of the set of matches decrypted input.

There are a few different ways to prevent our side-channel attack. One would be to implement a padding that will change the packets observation by a random increment of the packet lengths, which in turn render our attack useless. Another similar procedure would be to add data to the packet length so every packet is sent with a fixed size, i.e. setting the size for every packet to the largest possible packet size. However, as described in Section 2.5, there are ways to get around this, by implementing statistical prediction methods. There is also another approach that would render our attack useless. Since we are sending an amount of requests of polynomial magnitude, the host server, i.e Amazon UK's servers in this case, could employ a DDOS filter. As a consequence, the DDOS filter could trigger, for example, a blocking by IP and our profiles could not be generated. There are also ways to get around DDOS filters, such as IP-spoofing [15], however such methods are beyond the scope of this project.

7 Conclusion

In this project we have successfully performed a side-channel profiling attack against one of the largest e-commerce sites, Amazon UK, to decrypt encrypted web-search queries made in Amazon UK's main search field. The work was based on that of Chen et al. [2], with the difference that our target was a very large e-commerce website. Furthermore, we investigated the accuracy and efficiency that could be achieved through this attack - something that we have not found any data on in literature.

We created a framework that supports two methods for creating profiles that are used to decrypt search queries: *pre-computed* profiled and *on-the-fly* profiles. We evaluated the efficiency of these two methods in comparison to each other and found that on-the-fly profiles are preferable in normal environments where there is not a large amount of search traffic to decrypt. In highly active networks, however, on-the-fly profiles would probably bring too much overhead, and precomputed profiles would be preferable since they have near-to instant look-up. However, we also found that the accuracy of pre-computed profiles degrades quickly. Our results show that the validity could degrade with 90% in 48 hours.

In evaluating accuracy of this type of attack, we found that the amount of valid input (input that can be decrypted) is large for input of low sizes, and quite low for input of larger sizes. Furthermore, we found that the prediction rate is quite low for input of small sizes, but increases for input of larger sizes; we can conclude that the reason for this is that longer words can more easily be identified due to reduction by sequence, i.e. their character structure have a more unique sequence. Our results are based on a low sample size, and we cannot claim statistical significance. However, the purpose of this project was not to statistically determine accuracy and efficiency, but rather to give an idea of what can be expected for this type of attack.

We also discussed methods for mitigation and prevention of the kind of attack that was performed, and the limitations of our implementation. We can conclude there are several prevention mechanisms that would render our approach useless. However, it works well enough to break encryption on the world's largest e-commerce site, using a personal computer.

Bibliography

[1] A. Schaub, E. Schneider, A. Hollender, V. Calasans, L. Jolie, R. Touillon, A. Heuser, S. Guilley, and O. Rioul, “Attacking suggest boxes in web applications over https using side-channel stochastic algorithms,” in *Risks and Security of Internet and Systems*. Springer, 2014, pp. 116–130.

[2] S. Chen, R. Wang, X. Wang, and K. Zhang, “Side-channel leaks in web applications: a reality today, a challenge tomorrow,” in *Security and Privacy (SP), 2010 IEEE Symposium on*, ser. CCS ’10. New York, NY, USA: IEEE Computer Society, 2010, pp. 191–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=1849974

[3] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, “Automated detection and quantification of side-channel leaks in web application development,” in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS ’10. New York, NY, USA: ACM, 2010, pp. 595–606. [Online]. Available: http://dl.acm.org/citation.cfm?id=1866374

[4] C. O’Connor, “Wal-mart vs. amazon: World’s biggest e-commerce battle could boil down to vegetables - forbes,” http://www.forbes.com/sites/clareoconnor/2013/04/23/wal-mart-vs-amazon-worlds-biggest-e-commerce-battle-could-boil-down-to-vegetables/#541c88045b71, April 2013, (Accessed on 05/28/2016).

[5] Y. Zhou and D. Feng, “Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing,” in *IACR Eprint archive*, 2005, zyb@is.iscas.ac.cn 13083 received 27 Oct 2005. [Online]. Available: http://eprint.iacr.org/2005/388

[6] M. Backes, G. Doychev, and B. Köpf, “Preventing side-channel leaks in web traffic: A formal approach,” in *20th Annual Network and Dis- tributed System Security Symposium, NDSS 2013*, February 2013.

[7] S. Kent and K. Seo, “Security architecture for the internet protocol,” Internet Requests for Comments, BBN Technologies, RFC 4301, December 2005. [Online]. Available: https://tools.ietf.org/html/rfc4301

[8] F. Kurose and W. Ross, *Computer Networking: a top-down approch*, 6th ed. New Jersey: Pearson Education, Inc, 2013.

[9] S. A. Sharma and B. L. Menezes, "Implementing side-channel attacks on suggest boxes in web applications," in *Proceedings of the First International Conference on Security of Internet of Things*, ser. SecurIT '12. New York, NY, USA: ACM, 2012, pp. 57–62. [Online]. Available: http://doi.acm.org/10.1145/2490428.2490436

[10] M. Liberatore and B. N. Levine, "Inferring the source of encrypted http connections," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 255–263. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180437

[11] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. Berkeley, CA, USA: USENIX Association, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251327.1251352

[12] IANA, "Service name and transport protocol port number registry," www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=HTTPS, (Accessed on 05/27/2016).

[13] httpwatch.com, "Http chunked encoding | httpwatch," https://www.httpwatch.com/httpgallery/chunked/, (Accessed on 05/27/2016).

[14] stackoverflow.com, "javascript - prevent browser caching of jquery ajax call result - stack overflow," http://stackoverflow.com/questions/367786/prevent-browser-caching-of-jquery-ajax-call-result, (Accessed on 05/27/2016).

[15] H. Wang, C. Jin, and K. G. Shin, "Defense against spoofed ip traffic using hop-count filtering," *IEEE/ACM Trans. Netw.*, vol. 15, no. 1, pp. 40–53, Feb. 2007. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.890133