Attacking suggest boxes in web applications over https using stochastic side-channel algorithms

Alexander Schaub, Emmanuel Schneider, Alexandros Hollender, Vinicius Calasans, Laurent Jolie, Robin Touillon, Annelie Heuser, Sylvain Guilley, Olivier Rioul

Attacking Suggest Boxes in Web Applications Over HTTPS Using Side-Channel Stochastic Algorithms

Alexander Schaub[1], Emmanuel Schneider[1], Alexandros Hollender[1], Vinicius Calasans[1], Laurent Jolie[1], Robin Touillon[1], Annelie Heuser[2], Sylvain Guilley[2,3], and Olivier Rioul[1,2(✉)]

[1] Ecole Polytechnique, CMAP, Palaiseau, France
{alexander.schaub,emmanuel.schneider,alexandros.hollender,
vinicius.calasans,laurent.jolie,robin.touillon,
olivier.rioul}@polytechnique.edu
[2] Department Comelec, Télécom ParisTech, CNRS LTCI, Paris, France
{annelie.heuser,sylvain.guilley,olivier.rioul}@telecom-paristech.fr
[3] Secure-IC S.A.S., Rennes, France

Abstract. Web applications are subject to several types of attacks. In particular, side-channel attacks consist in performing a statistical analysis of the web traffic to gain sensitive information about a client. In this paper, we investigate how side-channel leaks can be used on search engines such as *Google* or *Bing* to retrieve the client's search query. In contrast to previous works, due to payload randomization and compression, it is not always possible to uniquely map a search query to a web traffic signature and hence stochastic algorithms must be used. They yield, for the French language, an exact recovery of search word in more than 30 % of the cases. Finally, we present some methods to mitigate such side-channel leaks.

Keywords: Side-channel leak · Web application · Suggest box · Stochastic algorithms · HTTPS

1 Introduction

Recent revelations by Edward Snowden have shown that there is no more privacy over the Internet. While it should perhaps not come as a surprise that governments worldwide are able to spy on their citizens, it has been widely believed that today's technology can at least protect our sensitive data from our neighbors or competitors. Actually, this is not so sure.

Search histories can be considered as sensitive data. As shown in the 2006 New York Times article [1], using a leakage in AOL search data, it is possible

The online demo of the attack (presented at the CRiSIS 2014 conference) is available on YouTube, at address: http://youtu.be/ynG6tuqeIuM.

Annelie Heuser is Google European fellow in the field of privacy and is partially founded by this fellowship.

to identify a person only from his search history. At company scale, a look at the search history of competitors can be used to predict their future actions in order to gain a strategical advantage over them. *Google* recognized the necessity to protect this sensitive data during the fall of 2011, when they announced that they enabled SSL for all of their signed-in users [2]. Later they forced the use of SSL for every search query, by automatically redirecting every user to the HTTPS version of their website [3]. Unfortunately, this is not enough to hide search queries completely because some information still leaks through side-channels.

Side-channel leaks appear each time an interaction between a user and a website requires transmission of information packets containing relevant data. Assuming that the connection between the client and the server is encrypted (using a protocol such as HTTPS), three parameters of the packet flow can be observed:

- *lengths* of individual packets;
- *directions* of packet flow (client to server or server to client);
- *times* of packets' departure and arrival [4].

By analyzing the packet flow associated with the suggest boxes[1] from *Google* (or any other search engine), it is observed that for every character typed in the search box, several packets are exchanged between the server and the client. One of these packets contains relevant data that depends on the list of suggestions from the search engine. In particular, by analyzing packet lengths associated to different characters, it is possible to guess the most likely word that the user typed in, and thus uncover sensitive information about his search history.

The remainder of this paper is organized as follows. First, Sect. 2 gives a current state of the art and Sect. 3 describes the structure of relevant information packets for today's *Google* and *Bing* (Microsoft) search engines. Then, Sect. 4 investigates novel algorithms to carry out side-channel attacks, that use data structures such as trees and stacks. The corresponding test results and some implementation issues are given in Sect. 5. Finally, Sect. 6 concludes by giving some perspectives on this work and methods for mitigating side-channel leaks.

2 Previous Work

Numerous studies on the detection and analysis of side-channel data leaks in web applications can be found in the literature. The general approach is to examine the properties of packet sequences sent between a client and a server, in order to infer a relationship between these properties and the exchanged information.

The authors of [5] used a deterministic model of web applications that allowed them to deduce recorded diseases and types of physician of users of some medical advice application, or to obtain details on the annual income and expenses of

[1] See description of *Google Instant*: http://goo.gl/WI9Zu and *Google Autocomplete*: http://goo.gl/jv3fQ.

a family in a tax return software used in the USA. These results were obtained through a simple analysis of packet sizes exchanged between the client and the server. Most of the time, one could map each input mouse selection or typed word to a single sequence of packet lengths. Therefore, the user input can be retrieved simply by comparing the sequence to a database of precomputed sequences.

In [6], the authors attempt to find the sources of a certain user connection by comparing the received data to a list of predetermined website profiles. Effective methods carry out the comparison on packet sizes, using either a similarity metric (Jaccard coefficient) or a Bayesian classification. Under certain assumptions, the origin of the data can be traced in more than 6 cases out of 10. The effectivity of this fingerprinting attack is improved in [7] using a multinomial Naïve-Bayes classifier.

An interesting information theoretic approach is investigated in [8] to describe the interaction between server and client. A web application is modeled as a finite-state machine, where state changes produce "traces" (specifically, the exchanged packets). Since these do not follow a deterministic law, a stochastic analysis is performed using mutual information to estimate the average reduction of uncertainty on the input when the attacker intercepts the packets. The method is tested on a simple yes/no questionnaire that redirects to two different sites depending on the answer.

In [9], side-channel attacks are carried out on search engines such as *Google*. The search box operates using AJAX to display suggestions to the client as he types search terms, and the attack again consists of intercepting the exchanged packets in order to infer the user's query. The authors have assumed a *deterministic* relationship between input letters and exchanged packet lengths[2]. The query can therefore be deduced by pre-computing every possible query and then comparing the captured packets using this information. While there may be several possible results for a same sequence of packet sizes, words that are not in some dictionary are unlikely to have been typed in. Therefore, it is only necessary to compute and store the sequences of packet sizes corresponding to legitimate words in the chosen dictionary.

We found that their method does not work any longer on *Google* since the suggestion list sent to the user has been changed in summer 2012 in such a way that there are now many possible sequences of packet lengths corresponding to a given search query.

3 Packet Structure

Exchanged packets between a client and server can be observed using an internet packet sniffer such as *Wireshark*. In order to determine which packet contains the relevant information, we simply decrypted the packet flow using Fiddler[3]

[2] More precisely, the sizes of the packets sent by the user are fixed for a given number of letters, and the sizes of received packets containing suggestions depend only on the word typed by the user (it may only change if *Google* changes the suggested search queries).

[3] http://www.telerik.com/fiddler.

and determined the size of the packet we were supposed to observe. After these initial tries, we were able, whenever a character was typed in, to filter out the only packet with a reasonable size. It is then easy to isolate the important packet containing the suggest-box data, as shown in Fig. 1.

We observed that packet sizes fluctuate for identical requests. To understand how, we have decrypted and unzipped the packets to study their structure. Even if the attacker will eventually not access the content of the encrypted packets, this structure helps understand how packet sizes and search queries are related.

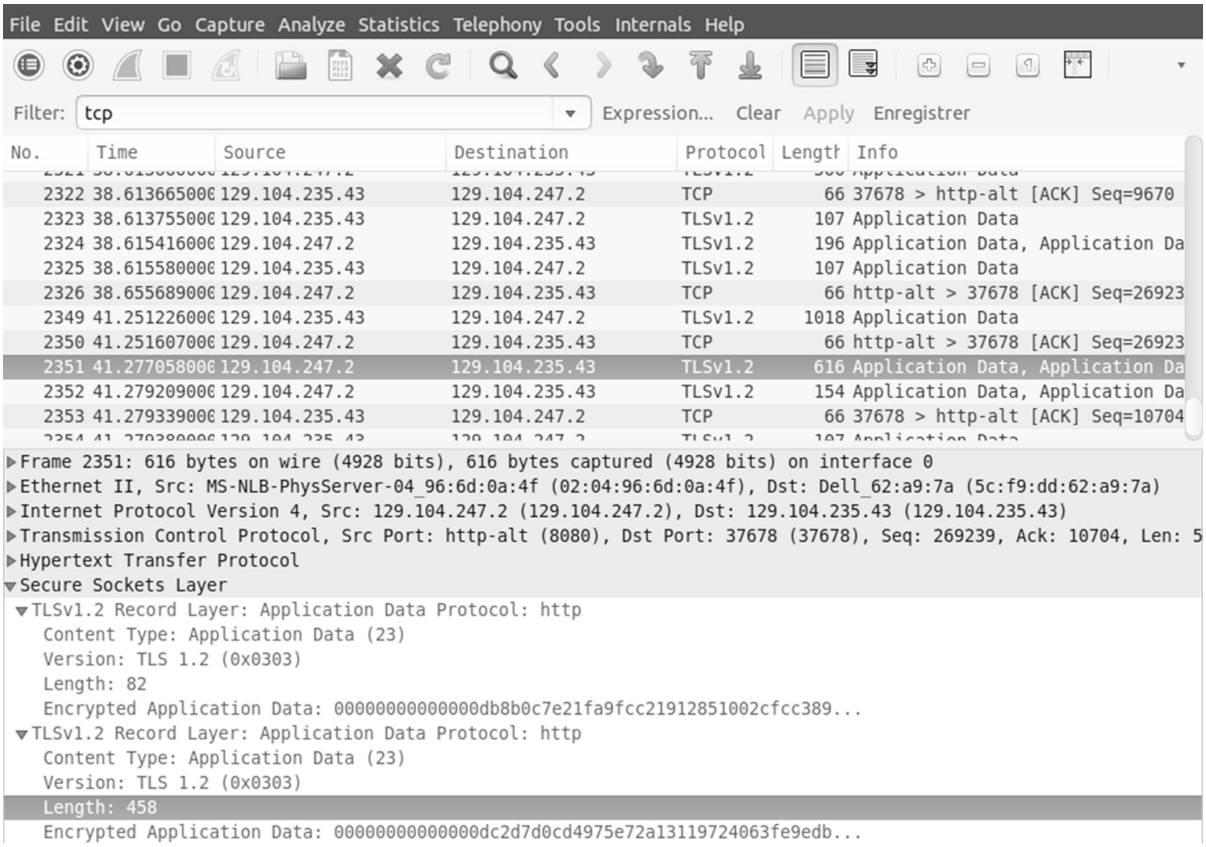

Fig. 1. A captured packet containing the suggest box data.

3.1 Google Packets

Previous works implementing side-channel attacks on suggest boxes did not analyze the structure of the exchanged packets. In [9], the only information needed is the link giving access to the packet related to a given search string. This is because, prior to summer 2012, there was no randomness in the packets that *Google* sent. Typing in an "a" for example, would always yield the same packet, and therefore the same packet length.

But at present, *Google* packets contain some kind of token (the value of which appears random to us), a milli-timestamp, and other numbers (which also appear random for an observer not aware of Google's protocols semantic). Figure 2 shows

```
{"e":"0k6DU5HUDfD50gW6_oCADg","c":0,"u":"https://www.google.fr/s
    ?gs_rn=45&gs_ri=psy-ab&pq=a&cp=1&gs_id=ds&xhr=t&q=a&
    es_nrs=true&pf=p&output=search&sclient=psy-ab&oq=&gs_l=&
    pbx=1&bav=on.2,or.r_qf.&bvm=bv.67720277,d.d2k&fp=
    1811953923e3f22&biw=1855&bih=718&tch=1&ech=2&psi=
    xk6DU-joOsaH0AXH-4HYAw.1401114308196.3","p":true,"d":"["a",[["a
    <b>mazon</b>",0],["a<b>llocine</b>",0],["a<b>meli</b
    >",0],["a<b>ir france</b>",0]],{"t":{"bpc":false,"tlw":
    false},"q":"_LZt_R7tHgjpU3Eask82JbvHZEY","j":"ds"}]"}/*""*/
```


Fig. 2. A packet sent by *Google* to a French user that hit an "a".

boxed elements that are random and change between two requests, even if the same list of suggestions is sent to the client. Since packets are compressed using *GZip*, the packet length also becomes random. For example, typing in "a" twice will yield different packet lengths.

Using this knowledge of packet structure it is possible to carry out a calculation in order to approximate the probability distributions of packet sizes for a given search string. First, as in [9], by using *Firefox*'s development tools we can identify a URL corresponding to the list of suggestions. At the time this paper was written (May 2014), it looked like:

```
url(search-string) = https://www.google.fr/s?gs_rn=45&gs_ri=psy-ab&
pq=a&cp=1&gs_id=ds&xhr=t&q=search-string&
es_nrs=true&pf=p&output=search&sclient=psy-ab&oq=&gs_l=&
pbx=1&bav=on.2,or.r_qf.&bvm=bv.67720277,d.d2k&fp=
1811953923e3f22&biw=1855&bih=718&tch=1&ech=2&psi=
xk6DU-joOsaH0AXH-4HYAw.1401114308196.3
```

From this we can approximate the required probability distribution as follows. First, the file given by `url(search-string)` is fetched. This is done only once for a given search string. Then, the identified random parts are replaced with randomly generated strings or numbers. Finally, the file is compressed using *GZip*, and the size of the compressed file is recorded. Repeating the last two steps (replace & compress) enables one to reliably estimate the distribution of the packet sizes, such as the one shown in Fig. 3. For our test purposes, we fetched every relevant file once, and replaced the random parts 1000 times in order to compute these probability laws.

3.2 Bing

Bing does not encrypt its traffic, but it is still interesting to analyze its auto-suggest feature. For example, side-channel attacks can also work on WPA-protected wireless traffic.

Before May 2014, the packets sent by Bing for the auto-suggest feature were neither compressed nor did they contain any random element. It was then very easy to find the search string by analyzing a sequence of packets: the same

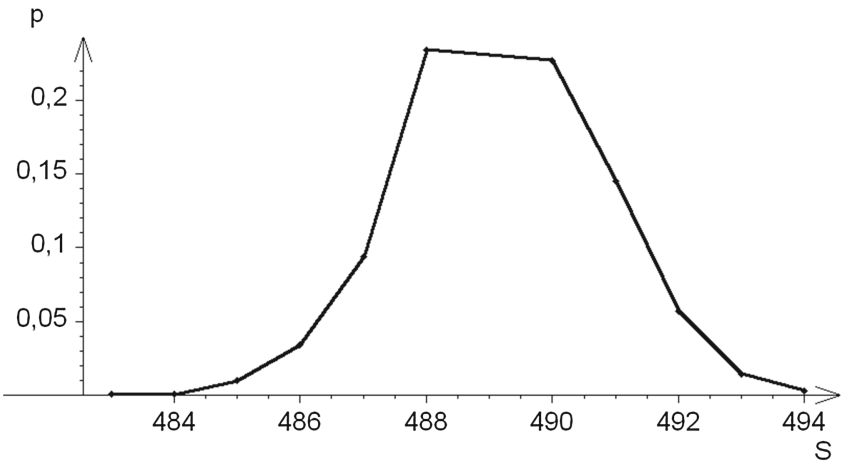

Fig. 3. Distribution of packet sizes (in bytes) for the letter "p", on the French version of *Google*.

method used for *Google* in [9] did work out very well. But now the situation has just changed: the packets are compressed and do contain some random elements. The corresponding file can be fetched at the following addresses:

```
url(search-string) = http://www.bing.com/AS/Suggestions?
pt=Page.Home&qry=search-string&cp=1&o=hs&css=1&
cvid=fbeb395a6a9b4f15bac899892c09b6a1
```

or

```
url(search-string) = http://www.bing.com/AS/Suggestions?
pt=Page.Home&qry=search-string&cp=1&o=hs&
cvid=fbeb395a6a9b4f15bac899892c09b6a1
```

There is an important difference between these two links: by specifying `css=1`, the whole CSS-code used for formatting the results will be sent. This happens when the first letter is typed in after having reloaded the web page. As a result, for some search strings, in particular those of length 1 (i.e., "a", "b", etc.), two different distributions must be computed by the attacker. However by looking at the file size it is easy to differentiate between a packet containing the CSS and a packet without CSS code (typically $\leq 3\,\mathrm{KB}$ for the CSS-free uncompressed version, $\geq 5\,\mathrm{KB}$ for the uncompressed file containing the CSS code).

4 Stochastic Algorithms

In this section, we describe the algorithms and data structures that we have used to solve the following problem.

Let l be the number of characters of a given word typed in by the user and let I be an interception vector containing the lengths of the l intercepted packets corresponding to the prefixes of length i of the given word for $i = 1$ to l. Using pre-computed probability distributions of packets lengths, determined as explained in the previous section, the goal is to find the most probable word (or list of words) that is most likely to have been typed by the user.

4.1 Restricting Possibilities

To simplify the problem, we make the plausible assumption that the user does not type a random sequence of letters but rather a sequence that makes sense. Therefore, we restrict the set of possible words to a certain "language" or dictionary, i.e., some predefined set of valid words. In our studies, we have chosen a simple French dictionary.

Restricting the set of possibilities has two main advantages. First, the algorithms will always return a valid word (or a list of valid words). Second, they will not waste computation time and memory space on words that do not even exist. As an example, there are about 11 million 5-letter sequences, for only 6812 valid French 5-letter words.

4.2 Data Structure

Once the dictionary is chosen, an adequate data structure representation of it should be implemented. Because a packet is intercepted for each prefix of the typed word, we choose to represent the set of all possible words of a given length l as a *prefix tree*. This tree has the empty word " " at its root, and contains each valid word of length l as a leaf. Going from the root to a certain word, one passes through the nodes representing all increasing prefixes of the word. This is called a *Trie* structure [10].

4.3 A Stack Algorithm

Recall from the previous section that at our disposal we have an algorithm, that we call LAW, that estimates the probability law of the packet length associated with a certain word prefix. Thus as an example, $LAW(\text{“plage”}, 435, \epsilon)$ returns an estimate of the probability that the packet sent by the server after the user has typed the last character of "plage", has a length in the interval $[\![435-\epsilon, 435+\epsilon]\!]$. Here ϵ is a tolerance parameter that is necessary for practical reasons. Because of the way the information is encapsulated during a packet exchange between a client and a server, it is not always possible to precisely determine the size of the relevant information that is hidden in the captured packets. An error of one or two bytes is not uncommon, and this is what ϵ represents.

Our first algorithm computes the likelihood f as the product of the estimated probabilities. For example, to measure how likely the prefix "pla" would be, we compute:

$$f(\text{“pla”}, I, \epsilon) := LAW(\text{“p”}, I[1], \epsilon) \times LAW(\text{“pl”}, I[2], \epsilon) \times LAW(\text{“pla”}, I[3], \epsilon)$$

where $I[i]$ is the size if the i-th intercepted packet. We have also tried other measures of likelihood f: sum of the prefix probabilities; or weighted sum (e.g., to emphasize the first letters of the word).

The detailed "stack" algorithm works as follows in the case $l = 5$ (`p.children` is the list of all children of prefix `p` in the prefix tree):

```
partial_solutions = {""}  # contains the empty word
amount_stored = {10, 20, 30, 20, 15} # example

for i = 1 to 5:
    new_solutions = {} # empty list
    for each prefix p in partial_solutions
        for each prefix r in p.children
            add r in new_solutions
    sort prefixes pr in new_solutions by value of f(pr, I, epsilon)
    put amount_stored[i] first prefixes from new_solutions into partial_solutions

return partial_solutions
```

At each step the algorithm keeps the best prefixes in a stack. It then goes deeper in the tree to find the best possible ways to extend those prefixes. The output is a list of words sorted by value of f, which are deemed most likely by the algorithm. Results obtained with this algorithm are presented in the next section.

4.4 Threshold Variant

A slightly modified version of the stack algorithm uses a different criterion to decide whether to keep or discard a prefix in the stack. Instead of selecting a fixed amount of prefixes in each step, all prefixes $\mathtt{pr}$ for which the value $LAW(\mathtt{pr}, I, \epsilon)$ is greater than a given threshold are kept. The value T of this threshold varies from one step to another. Only the "local" probability $LAW(\mathtt{pr}, I, \epsilon)$ is taken into account in each step, not the global $f(\mathtt{pr}, I, \epsilon)$, resulting in a more efficient computation. Results obtained with this variant are also presented next.

5 Test Results

This section presents the results of our algorithms tested on *Google*, by simulating an interception over Ethernet or Wifi. To simplify we assume a fixed value $l = 5$, i.e., a 5-letter French word is typed by the user.

5.1 Results Using the Stack Algorithm

The number of stored prefixes in each step from $i = 1$ to 5 were chosen as $\{20, 30, 50, 30, 15\}$. The final list will thus contain 15 possible words, ranked from the most to the least probable. Ten different target words were chosen, with ten retries per target, yielding 100 result samples. Table 1 shows the rank $\in [\![1, 15]\!]$ of the target word in the final list or a cross ($\times$) if the word was not found at all.

Table 2 shows that much poorer results would be obtained if one kept only 15 prefixes at *each* step in the algorithm. This shows the importance of considering larger numbers of stored prefixes at intermediate steps.

Table 1. Results of the stack algorithm when f is the product of the probabilities: $f = \prod_i LAW(pr_i, I[i], \epsilon)$. Success rates are **81 %** for target found in the final list; **52 %** found in the top 3; **34 %** ranked first.

Tested word	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
bases	4	1	×	2	4	8	×	×	6	9
barbe	2	3	1	2	5	7	4	×	1	4
bague	1	1	×	1	1	1	6	1	1	1
atome	1	×	1	1	3	3	2	6	11	4
cache	15	7	×	×	12	7	×	×	11	×
cadre	15	×	×	4	×	15	×	×	×	×
maman	1	2	1	2	1	1	1	1	1	1
parle	1	2	1	1	2	10	5	5	2	1
pomme	5	1	8	3	1	2	1	1	2	9
neige	2	5	1	×	1	10	1	1	3	2

Table 2. Results of the stack algorithm when f is the product of the probabilities and only 15 prefixes are kept at each step in the algorithm. Success rates drop down to **50 %** for target found in the final list; **38 %** found in the top 3; **25 %** ranked first.

Tested word	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
bases	1	×	×	1	×	×	1	×	×	1
barbe	×	2	×	1	×	×	×	×	3	×
bague	×	1	×	×	×	×	2	×	1	×
atome	2	2	×	×	×	1	1	×	×	3
cache	×	×	9	10	11	10	11	×	×	×
cadre	×	6	×	×	×	4	×	×	3	×
maman	1	2	×	1	×	1	1	×	1	×
parle	7	1	×	×	1	1	×	1	1	2
pomme	5	2	5	×	1	2	5	3	×	1
neige	×	2	1	×	1	1	×	1	5	×

5.2 Other Choices for the Likelihood Function

For the stack algorithm with a fixed number of kept words at each step, in addition to the choice where f is the product of the probabilities:

$$f = \prod_i LAW(pr_i, I[i], \epsilon)$$

we have tested other formulas for the likelihood function: sum of the probabilities:

$$f = \sum_i LAW(pr_i, I[i], \epsilon)$$

and weighted sum

$$f = \sum_i (n-i) LAW(pr_i, I[i], \epsilon)$$

that gives more importance to the first letters. The choice of likelihood as a product of probabilities gives the best results among the tested functions (see Fig. 4 below), which is coherent with the theory.

5.3 Results Using the Threshold Variant

For the threshold version of the algorithm, taking likelihood f as the product of the probabilities is again the best choice. A lower threshold allows more accuracy,

Table 3. Results of the stack algorithm with threshold $T = 0.1$. Success rates are **89 %** for target found in the final list; **56 %** found in the top 3; **36 %** ranked first.

Tested word	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
bases	1	×	×	×	1	1	2	2	2	1
barbe	1	1	1	1	2	×	2	2	×	1
bague	6	1	1	7	1	5	1	1	3	1
atome	1	2	1	5	3	1	×	1	1	1
cache	4	17	11	4	11	37	18	21	4	20
cadre	30	11	29	18	26	56	13	3	3	8
maman	1	1	1	1	1	1	1	1	1	1
parle	2	3	1	×	3	4	2	3	×	4
pomme	5	4	4	1	×	1	5	1	2	5
neige	2	4	1	3	6	3	4	5	×	×

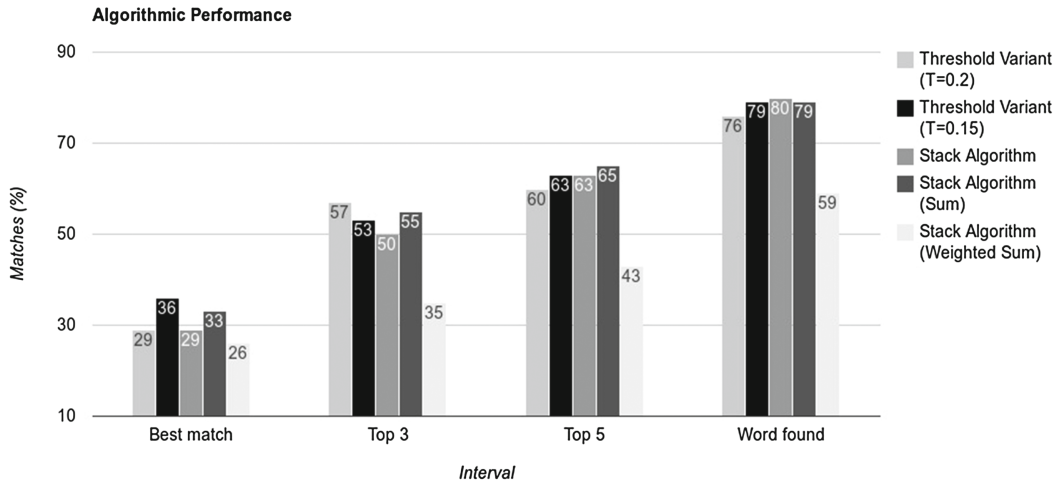

Fig. 4. Results of additional test runs. The threshold variant was tested once with a threshold of 0.15 and once with a threshold of 0.2. The likelihood variants were the product, sum and weighted sum of the probabilities.

in spite of a slightly longer execution time (which remains less than 15 min). Table 3 is presented similarly as above, except that the target rank may now be larger than 15.

5.4 Global Performance

Our results are summarized in Fig. 4. This chart shows how often the target word was found by the algorithm, how often it was among the best 5 matches, among the best 3 matches and how often it was the best match.

The results show that the variants perform similarly, except the weighted sum version which actually performs worse. On average, the target word is in the word list 8 times out of 10, in the best three matches more than 5 times out of 10, and is the best match about 3 times out of 10.

Interestingly, the tables show that some words are missed quite often, like *cache* or *cadre*. We found two plausible reasons for this:

- *Google* loads the result page after three ("cad") of four ("cadr") letters. Since we have assumed that two result page loadings cannot be distinguished, there remains few different packet sizes available;
- it turns out that those sizes are very common among all possible packet sizes (about 480 bytes which is the most probable packet length): too many words match the same sizes.

5.5 Implementation Issues

From our experience, the step that is always the most time-consuming is the first one that fetches the relevant file from the search engine. It is a good idea to cache the results in order to save time. Once a probability law is computed, it is stored so that it is not necessary to compute it again. This is particularly

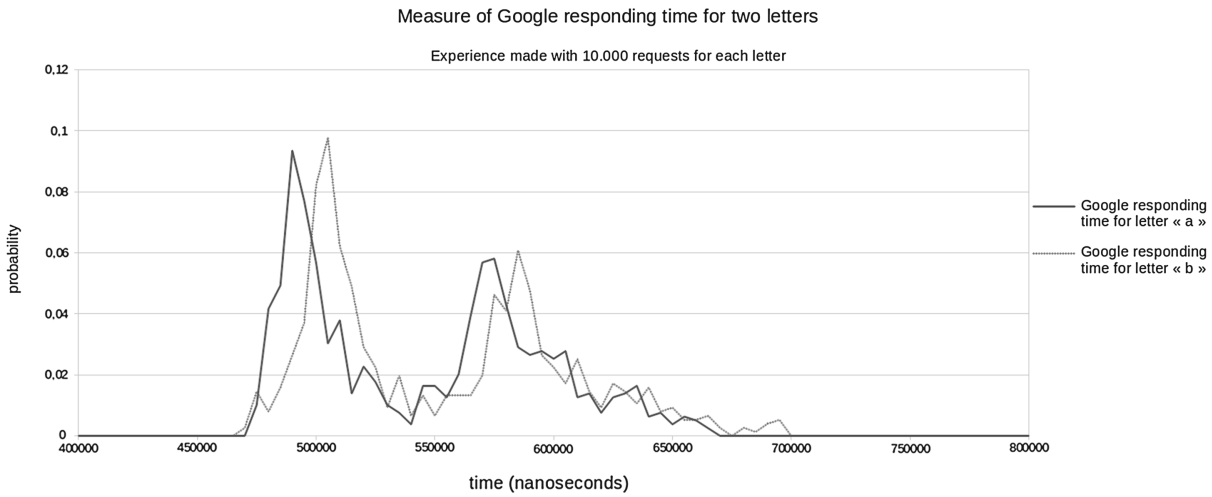

Fig. 5. Google response time for two different queries.

effective when several tests are performed; even on a single run, the duration may be divided by 2. Also, most of the time is wasted by waiting for the search engine to respond, using several *threads* can be more efficient.

Side-channel attacks can be used to work over Ethernet as well as protected Wifi networks. However, we noticed that *Google* often sends the important data in a packet containing two or more encrypted Application Data chunks. This is not a problem for an attack over Ethernet, since the different chunk sizes can be easily determined, but it is more of an issue over Wifi. Also, some constant *offset* is to be determined, that depends on the wireless access point configuration, which allows to convert the compressed suggestion data size to the actual captured packet size. This offset depends on the other data chunks in the intercepted packet, and it may therefore require some time to determine the actual suggestion data size for a packet captured over Wifi.

6 Conclusion

In this paper, the side-channel leakage of a major search engine, *Google*, has been analyzed. Knowledge of encrypted packet lengths can be used to deduce the user's search query, even if the packet sizes are randomized. Stack algorithms are presented to achieve this, based on multiple probabilities for each typed prefix and on natural language to limit the possibilities. These algorithms can be adapted to any other search engine that uses suggest boxes or similar features. Therefore, randomizing packet lengths is certainly not enough to mitigate side-channel leaks.

6.1 Perspectives

Some improvements and issues remain topics for future investigations.

Several words. In order to handle the use of the space key, it would be necessary to slightly alter the structure of the tree representing the dictionary. Every leaf (word) should be arrowed back to the root, where the arrow represents the whitespace character. It would actually not be a tree anymore, but rather a cyclic structure.

Use of backspaces. Our algorithm cannot find the search query if an user hits the backspace key because it would be searching for a word that would be too long. For example, for the word "mub←m", one would receive 5 packets related to the queries "m", "mu", "mub", "mu" and "mum". It is possible to add words like "mub←m" (considering this as a 5-letter word) but this increases the size of the 5-letter dictionary by $26^4 = 456976$ times the size of a 4-letter dictionary (even without considering that the backspace key may be used more than once).

Automatic downloads. Sometimes, *Google* is pretty sure about what the user is looking for and loads the corresponding page—for example, if one starts by

hitting "f", Google will load the page with *Facebook*-oriented links.[4] This of course results in many packets sent by *Google* which can be easily detected.

Localization and customization. *Google*'s suggestions depend on the user's language defined in his/her *Google* homepage, and on the country of his/her ISP. They also depend on the browsing history and previous search history. This is the major problem for our algorithm since the latter relies on the fact that the victim and attacker get the same suggestions from *Google*. This would still be the case, however, if the victim uses the "Private Mode" implemented on most browsers—which here, ironically, causes a loss of privacy. Our algorithms could also be tested with other dictionaries, for example with a complete English dictionary and English search-terms, to see how it performs in this case. We don't expect the results to be much different.

Server's response time. We have only considered the lengths of the packets that are being sent. Another important side-channel information would be the time when the packets arrive to the client (or are intercepted by the attacker [11]). Figure 5 shows the estimated probability of time between the departure of the request and the arrival of *Google*'s answer for two different letters "a" and "b". The two curves seem shifted: *Google*'s computation time for letter "b" is longer than the one for letter "a", and this type of information could have been used in our algorithm. However, the delay between the two signals is very small compared to their deviations. Also, computing these curves is quite time-consuming—unlike packet lengths, it is not possible to compute the response times after having fetched only one file.

Multiple requests. One possible improvement of the attack would be to make *Google* send the suggestions several times, since this would reduce the uncertainty of the packet lengths. This could perhaps be achieved by re-sending the victim's encrypted request to *Google*, but it may not be easy to trick *Google* into thinking that the attacker is the victim.

6.2 How to Mitigate Side-Channel Leaks

Today, as we have shown, using a simple personal computer, it is possible to spy on anybody using a Wifi connection, even if this connection is made secure by other means. This is a serious threat to privacy over the Internet. Even though the randomization of packet lengths makes it harder to infer a search query, it is still possible to guess the target correctly in many cases. There have been numerous attempts to mitigate side-channel leaks in general [12,13], but it is generally considered that preventing *every* side-channel leak source is very difficult [14].

However, for the particular leak exploited in this paper, it would be easy to implement an efficient countermeasure by sending only packets of a given, fixed size (e.g., the size of the longest possible packet in response of a request). A similar procedure can be carried out for response times. For example, the server could always wait a fixed time before answering.

[4] This is known as *Google Instant*.

The cost of such a procedure can be criticized, but it would definitely make our present method useless. Nonetheless, we notice that such method can be limited to sensitive traffic (e.g., contextual to user interaction with the server). A simple way of achieving this would be to pad every packet to a fixed size M, and disable any compression feature. The remaining problem is to choose the correct value of M. A solution would be to choose the maximum packet length for M, but it is not always possible to determine this maximum. Whenever the initial packet length exceeds M, one could pad it to the closest multiple of M. Although this gives the attacker some information, it should not be enough to guess the search query.

References

1. A Face Is Exposed for AOL Searcher, New York Times article, 9 August 2006. http://select.nytimes.com/gst/abstract.html?res=F10612FC345B0C7A8CDDA10894DE404482. Accessed 27 July 2014
2. Making Search More Secure, 18 October 2011. http://googleblog.blogspot.fr/2011/10/making-search-more-secure.html. Accessed 27 July 2014
3. Post-PRISM, Google Confirms Quietly Moving To Make All Searches Secure, Except For Ad Clicks, 23 September 2013. http://searchengineland.com/post-prism-google-secure-searches-172487. Accessed 17 July 2014
4. Cantino, A.: Demasking Google Users With a Timing Attack (blog post). http://blog.andrewcantino.com/blog/2014/09/04/demasking-google-users-with-a-timing-attack/
5. Chen, S., Wang, R., Wang, X., 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 (SP 2010), pp. 191–206 (2010)
6. Liberatore, M., Levine, N.B.: Inferring the source of encrypted HTTP connections. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006), pp. 255–263. ACM, New York (2006)
7. Herrmann, D., Wendolsky, R., Federrath, H.: Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial Naïve-Bayes classifier. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security (CCSW 2009), pp. 31–42 (2009)
8. Mather, L., Oswald, E.: Pinpointing side-channel information leaks in web applications. J. Cryptogr. Eng. **2**(3), 161–177 (2012). Also available in ICAR ePrint 2012:269
9. Sampreet Sharma, A., Bernard Menezes, M.: Implementing side-channel attacks on suggest boxes in web applications. In: Proceedings of the First International Conference on Security of Internet of Things, SecurIT 2012, Amritapuri, Kollam, pp. 57–62 (2012)
10. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
11. Tey, C.M., Gupta, P., Gao, D., Zhang, Y.: Keystroke timing analysis of on-the-fly web apps. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 405–413. Springer, Heidelberg (2013)
12. Nassar, M., Guilley, S., Danger, J.-L.: Formal analysis of the entropy/security trade-off in first-order masking countermeasures against side-channel attacks. In: Bernstein, D.J., Chatterjee, S. (eds.) INDOCRYPT 2011. LNCS, vol. 7107, pp. 22–39. Springer, Heidelberg (2011)

13. Backes, M., Doychev, G., Köpf, B.: Preventing side-channel leaks in web traffic: a formal approach. In: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, 24–27 February 2013, 17 p. http://internetsociety.org/doc/preventing-side-channel-leaks-web-traffic-formal-approach
14. Dyer, K.P., Coull, S.E., Ristenpart, T., Shrimpton, T.: Peek-a-Boo, i still see you: why efficient traffic analysis countermeasures fail. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP 2012), San Francisco, California, USA, pp. 332–346 (2012)