What Are You Searching For?
A Remote Keylogging Attack on Search Engine Autocomplete

John V. Monaco
Naval Postgraduate School, Monterey, CA

Abstract

Many search engines have an autocomplete feature that presents a list of suggested queries to the user as they type. Autocomplete induces network traffic from the client upon changes to the query in a web page. We describe a remote keylogging attack on search engine autocomplete. The attack integrates information leaked by three independent sources: the timing of keystrokes manifested in packet inter-arrival times, percent-encoded Space characters in a URL, and the static Huffman code used in HTTP2 header compression. While each source is a relatively weak predictor in its own right, combined, and by leveraging the relatively low entropy of English language, up to 15% of search queries are identified among a list of 50 hypothesis queries generated from a dictionary with over 12k words. The attack succeeds despite network traffic being encrypted. We demonstrate the attack on two popular search engines and discuss some countermeasures to mitigate attack success.

1 Introduction

Search queries contain sensitive information about individuals, such as political preferences, medical conditions, and personally identifiable information [7, 25]. They can reveal user demographics, hobbies, and interests, and are routinely used for targeted advertising [4, 24]. To protect user privacy, all major search engines now encrypt search query traffic.

Autocomplete is a feature that provides suggested queries to the user as they type based on the partially completed query, trending topics, and the user's search history [2]. Intended to enable the user to find information faster, autocomplete requires the user's client to communicate with the server as keyboard input events are detected. As a result, the user's keystrokes manifest in network traffic.

We present a remote keylogging attack on websites that implement autocomplete. The attack detects keystrokes in encrypted network traffic and identifies search queries using information from three independent sources: keystroke timings manifested in packet inter-arrival times, percent-encoding of Space characters in a URL, and the static Huffman code used in HTTP2 header compression.

The attack we developed, called KREEP (Keystroke Recognition and Entropy Elimination Program), consists of five stages: keystroke detection, which separates packets that correspond to keystrokes from background traffic; tokenization to delineate words in the packet sequence; dictionary pruning, which uses an HTTP2 header compression side channel to eliminate words from a large dictionary; word identification, performed by a neural network that predicts word probabilities from packet inter-arrival times; and a beam search, which generates hypothesis queries using a language model. KREEP is a remote passive attack that operates entirely on encrypted network traffic.

Autocomplete has been incorporated into almost every major search engine. We demonstrate our attack on two popular search engines and evaluate its performing using a collected dataset of 16k search queries. Using a dictionary with over 12k words, KREEP identifies 15% of queries and recovers up to 60% of the query text among a list of 50 hypothesis queries. The attack is robust to packet delay variation (PDV). We simulate up to ±32ms of network noise and find relatively little loss in performance with moderate levels of PDV. However, the attack is not robust to padding and we propose a simple padding defense that mitigates both the HTTP2 header compression side channel and ability to delineate words.

To summarize, the main contributions of this work include:

1) *A method to detect packets induced by autocomplete and delineate words in a query.* The pattern of autocomplete packet sizes from each search engine is characterized by a deterministic finite automaton (DFA). We generalize the longest increasing subsequence problem, which has an efficient dynamic programming solution, to that of finding the longest subsequence accepted by the DFA. This approach can detect keystrokes in network traffic with near-perfect accuracy and delineate words with greater than 90% accuracy.

2) *A side channel attack that leverages the static Huffman code used in HPACK, the HTTP2 header compression for-*

mat. Previously, it was shown that HPACK leaked relatively little information through compressed size [50]. However, with autocomplete, a search query is built up incrementally one character at a time and then recompressed. Due to this incremental compression, the information leaked is more than previously thought. We describe a method to leverage this information leakage to prune a dictionary, which increases the accuracy of our remote keylogging attack.

3) *A neural network that identifies words from keystroke timings.* We define a neural network architecture that takes into account the preceding and succeeding context of each observed timing and a method to identify words from a dictionary containing over 12k entries. The network is trained on keystrokes recorded from 83k typists, and words are correctly identified with 19% accuracy.

4) *The integration of a language model and keystroke timing attack to leverage the relatively low entropy of English language.* Previous keystroke timing attacks have noted the relatively low entropy of natural language compared to password input [47]. We introduce a method that combines a keystroke timing attack with a language model to generate hypothesis search queries. The use of a language model significantly improves performance.

In the next section, we provide background information on keylogging side channels and autocomplete. The attack workflow and threat model are described in Section 3, followed by keystroke detection and tokenization in Section 4. Dictionary pruning and the HTTP2 header compression side channel are described in Section 5. Word identification from timings and the language model are described in Section 6. Sections 7 and 8 contain results and discussion, respectively, and Section 9 concludes.

2 Background

2.1 Keylogging side channels

A keylogging side channel attack aims to recover the keystrokes of a victim through unintended information leakage. Such attacks have been demonstrated for a wide range of modalities such as acoustics [5], seismic activity [31], hand motion [54], and spikes in CPU load [46]. These generally fall into two different categories: spatial attacks, which utilize a channel that leaks spatial information about where a key is located on the keyboard, and temporal attacks, which utilize a channel that leaks only the timing of the keyboard events [34]. Our attack leverages both spatial and temporal information leaked through network traffic generated by a website with autocomplete.

Temporal keylogging attacks attempt to recognize which keys a user typed based only on the key press and release timings. This is possible because different key sequences can result in characteristic time intervals, such as typing the key sequence “th” quicker than “aq”. Consequently, the exposure of keyboard event timings is a threat to user privacy. Remote keystroke timing attacks may target applications in which a keystroke induces network traffic from the victim’s host, such as SSH [47] or a search engine with autocomplete functionality [51]. Packet inter-arrival times, when observed remotely, reveal the time between successive keystrokes. Keyboard input events can also be detected from within a sandboxed environment on the host [46] or on a multi-user system [59].

Keylogging attacks can be characterized by the type of input that occurs. For password input, an attack may assume that each key has an equal probability of occurrence, i.e., maximum entropy, whereas for natural language it is often assumed that the user typed a word contained in a dictionary [29]. For the purpose of identifying search queries, we assume natural language input which enables KREEP to leverage a language model in generating hypothesis queries.

Two main problems arise when trying to determine keystrokes from timings. The first is keystroke detection: given a sequence of events, such as network packets, spikes in CPU load, or memory accesses, determine which events correspond to keystrokes and which do not. This is a binary classification problem. In our attack, we consider a sequence of network packets emitted by the victim which includes background traffic in addition to the HTTP requests induced by autocomplete. The second problem is key identification: given that a key press has occurred, the attacker must determine which key it was. This is a multi-class classification problem. In our attack, we assume that each key is either an English alphabetic character (A-Z) or the Space key, for a total of 27 keys.

We address the problems of keystroke detection and key identification separately. KREEP detects keystrokes by finding a subsequence of packet sizes that are characteristic of autocomplete requests. For key identification, KREEP leverages both packet size and packet inter-arrival timings, which faithfully preserve key-press latency.

2.2 Web search autocomplete

Many websites have autocomplete functionality. With this feature, a list of suggested search queries is presented to the user as they enter text into a search form. The list of suggested queries is determined by an algorithm based on the user’s search history, current trending topics, and geographic location [2]. Because the suggestions are automated, this can sometimes result in unfavorable associations implied between search terms which has made autocomplete the focus of several legal disputes [27].

As changes to the query are detected, the client sends an HTTP GET request to the server and the server responds with a list of suggested search queries [26]. This results in a series of HTTP requests following keyboard events, such as those shown in Figure 1. The request contains the partially completed query in addition to other parameters, such as an

	Size	URL
Google	163	?q=**t**&cp=1&...
	164	?q=**th**&cp=2&...
	164	?q=**the**&cp=3&...
	166	?q=**the%20**&cp=4&...
	167	?q=**the%20l**&cp=5&...
	168	?q=**the%20la**&cp=6&...
	169	?q=**the%20laz**&cp=7&...
	170	?q=**the%20lazy**&cp=8&...
	172	?q=**the%20lazy%20**&cp=9&...
	173	?q=**the%20lazy%20d**&cp=10&...
	173	?q=**the%20lazy%20do**&cp=11&...
	174	?q=**the%20lazy%20dog**&cp=12&...

	Size	URL
Baidu	661	?wd=**t**&csor=1&...
	668	?wd=**th**&csor=2&pwd=t&...
	670	?wd=**the**&csor=3&pwd=th&...
	674	?wd=**the%20**&csor=4&pwd=the&...
	678	?wd=**the%20l**&csor=5&pwd=the%20&...
	680	?wd=**the%20la**&csor=6&pwd=the%20l&...
	682	?wd=**the%20laz**&csor=7&pwd=the%20la&...
	684	?wd=**the%20lazy**&csor=8&pwd=the%20laz&...
	688	?wd=**the%20lazy%20**&csor=9&pwd=the%20lazy&...
	693	?wd=**the%20lazy%20d**&csor=10&pwd=the%20lazy%20&...
	695	?wd=**the%20lazy%20do**&csor=11&pwd=the%20lazy%20d&...
	697	?wd=**the%20lazy%20dog**&csor=12&pwd=the%20lazy%20do&...

Figure 1: Autocomplete requests for the query "the lazy dog" in Google (left) and Baidu (right). After each key press, the client sends an HTTP GET request that contains the partially completed query in the URL (shown in bold). Packet size is in bytes.

authentication token and page load options, which generally do not change between successive requests. As a result, each request changes by only a single character, and the size of each packet increases by about 1 byte over the previous.

There are primarily two methods to implement autocomplete [35]. The first is a polling model in which a web page periodically checks the contents of the query input field at fixed intervals. When a change is detected, an autocomplete request is sent to the server to retrieve the query suggestions. Depending on the polling rate and the speed of the typist, an autocomplete request may not immediately follow every keystroke. If two keystrokes occur before the polling timer expires, then they will both be included in the next autocomplete request. In this situation when the typing rate exceeds the polling rate, the keyboard input event times are not faithfully preserved in packet inter-arrival times due to multiple keys being merged into a single request.

The second method of implementing autocomplete is a callback model in which the requests are triggered by HTML DOM `keydown` or `keyup` input events. In this approach, each autocomplete request immediately follows each input event such that the packet inter-arrival times faithfully preserve the time between keyboard events. Non-printable characters, such as Shift, Ctrl, and Alt, are ignored since these alone do not result in visible changes to the query.

We focus only on search engines that implement autocomplete requests triggered by `keydown` events. This results in packet inter-arrival times that are highly correlated with key-press latencies, i.e., time between successive `keydown` events. Previously, we determined that Bing implements a polling model with 100 ms timer, DuckDuckGo implements a callback model triggered by `keyup` events, and Baidu, Google, and Yandex implement a callback model triggered by `keydown` events [35]. Because Yandex is not vulnerable to the method of tokenization described in Section 4, we consider only search engines Google and Baidu. As of January, 2019, Google search comprises over 90% of worldwide market share [49], and Baidu comprises over 70% of the market share within China [48].

3 Attack overview

In this section, we define the threat model and describe the attack workflow. We then summarize the performance metrics used to evaluate each component of KREEP separately as well as overall attack success.

3.1 Threat model

We assume a remote passive adversary who can capture encrypted network traffic emitted by a victim using a search engine with autocomplete. We do not make any assumptions about background traffic or the ability to detect when a web page loaded; KREEP is able to isolate the subsequence of packets that contain autocomplete requests.

We assume the victim types only alphabetic keys and the Space key (27 keys total) to form a query made of lowercase English words with each word separated by a Space. This excludes queries that were copied and pasted, the use of Backspace and Delete keys, and any other input that might cause the cursor to change position, such as arrow keys. The victim might select an autocomplete suggestion before typing a complete query; KREEP can identify the query up to the point a selection was made.

The query must contain words in a large English dictionary known to the attacker. We use a dictionary of over 12k words comprised of the 10k most common English words [32] together with English words that appear in the Enron email corpus and English gigaword newswire corpus [19] (used to simulate search queries, see Section 7.1 for dataset details). KREEP does not require any labeled data from the victim for the keystroke timing attack; the neural network that performs

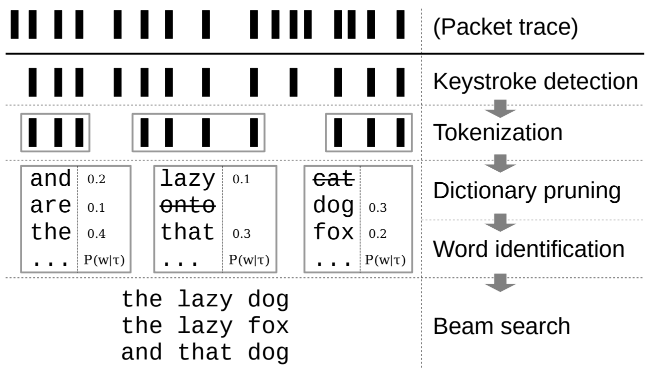

Figure 2: Attack workflow. Input to KREEP is a packet trace containing autocomplete and background traffic; output is a list of hypothesis search queries. Each component provides input to the next. See text for component definitions.

word identification is trained on an independent dataset. We assume that the attacker has access to this dataset.

3.2 Workflow

Our attack consists of five stages applied in a pipeline architecture shown in Figure 2 and summarized below.

Keystroke detection: packets that correspond to keyboard events are first detected from the full packet trace. This is a binary classification problem where each packet is labeled as either key-press or non-key-press. Each autocomplete request contains the query typed up to that point, so the sequence of autocomplete packet sizes has approximate linear growth over time. This makes it possible to separate keystrokes from background traffic, described in Section 4.2.

Tokenization: from the detected subsequence of packets, words are delineated based on packet size differences. Tokenization is also a binary classification problem where each packet is labeled as either Space or non-Space. Space characters in a URL are encoded by a three-byte escape sequence whereas other characters occupy a single byte. This behavior enables tokenization, described in Section 4.3.

Dictionary pruning: packet size differences are compared to a dictionary to eliminate words that could not have resulted in the observed sequence. This effectively prunes the hypothesis query search space. Dictionary pruning is possible due to the static Huffman code in HTTP2 header compression. This side channel is described in Section 5.

Word identification: the probability of each word remaining in the dictionary is determined from the observed packet inter-arrival times, which faithfully preserve key-press latencies. Word identification is performed by a neural network described in Section 6.1.

Beam search: word probabilities are combined with a language model in a beam search that generates hypothesis queries. The number of hypothesis queries is controlled by the beam width. The beam search is described in Section 6.2.

3.3 Performance metrics

We measure the performance of each component of the attack separately as well as overall attack success.

Both keystroke detection and tokenization are binary classification problems. For keystroke detection, a false positive occurs when a packet is incorrectly labeled as an autocomplete request, and a false negative occurs when an autocomplete request packet is missed. Likewise, a tokenization false positive occurs when a letter is incorrectly labeled as a Space, and false negative occurs when a Space is missed. Let fp, fn be the number of false positives and false negatives, and let tn, tp be the number of true negatives and true positives, respectively. We measure the performance of both keystroke detection and tokenization by the F-score,

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{1}$$

where

$$\text{Precision} = \frac{tp}{tp+fp}, \quad \text{Recall} = \frac{tp}{tp+fn} \ . \tag{2}$$

The F-score varies between 0, for missing all positives, and 1, for perfect precision and recall. Both keystroke detection and tokenization provide input to later stages of the attack, the success of which critically depends on performing well at both these tasks. As demonstrated in Section 7.2, making these tasks more difficult significantly reduces overall performance.

The utility of dictionary pruning is measured by the information gain due to incremental HTTP2 header compression. We compare this to the information gain in a classical compression side channel where only the total compressed size of the query is known.

For word identification, we report the word classification accuracy from packet timings, assuming perfect detection and tokenization. This evaluates word identification separately from the other components.

We consider two metrics to measure overall attack success. First is the rate at which a query is correctly identified among the list of hypothesis queries. Using a beam width of 50, this corresponds to a top-50 classification accuracy. Since the hypotheses may contain queries that are close, but do not exactly match, the true query, we also consider the Levenshtein edit distance between the true and hypothesis queries. Edit distance is used instead of character or word classification accuracy since failures in keystroke detection can result in a predicted query that is either shorter or longer than the original query. This metric is thought to better reflect the overall performance of a keylogging attack in such cases [16]. We report the minimum edit distance among the hypotheses to the true query, which roughly corresponds to the maximum proportion of keys that are correctly identified.

4 Keystroke detection and tokenization

In this section, we characterize the network traffic emitted by autocomplete in two different search engines. We then describe the first two stages of attack: a method to detect packets that contain autocomplete requests and a method to delineate words in the query. Both stages leverage characteristics of autocomplete packet sizes.

4.1 Autocomplete packet sizes

The problem of keystroke detection involves deciding whether each captured packet was induced by a keyboard event or not. As the user types a query into a search engine with autocomplete, the client emits HTTP requests that contain the partially completed query, such as those shown in Figure 1. However, these are mixed together with requests to load page assets, such as HTML and CSS files, AJAX requests supporting dynamic web content, and other background traffic. We found that typing a query with 12 characters on Google search induces 95 outgoing packets with payload greater than 0 bytes (436 packets including those with empty payloads), only 12 of which correspond to autocomplete requests.

Each autocomplete request contains a new character appended to the URL path. As a result, the sequence of packet sizes is monotonically increasing, shown in Figure 1. We perform keystroke detection by isolating a subsequence of packets that exhibit this pattern, taking into account the particular behavior of each search engine described below.

The behavior of each search engine is characterized by the sequence of size differences between successive autocomplete request packets. That is, let s_i be the size in bytes of the ith autocomplete request and s_0 the size of the first request. Packet size differences are given by $d_i = s_i - s_{i-1}$ for $i > 0$. This sequence reflects packet size growth as a function of query length, invariant to the size of other parameters contained in the request which vary across hosts due to different sized identifiers, authentication tokens, and page load options. However, these parameters typically remained unchanged in successive autocomplete requests from a single host.

Figure 3 shows the distribution of d_i as a function of query length for both Google and Baidu. From this figure and a manual inspection of several HTTP request packets, we make several observations about the behavior of each search engine. We then use these observations to build a DFA that accepts a sequence of autocomplete packet size differences.

Google autocomplete emits packets that typically increase by between 0 and 3 bytes. As each new character is appended to the “q=” parameter of the URL, the size increases by about a byte. The 2 and 3 byte increases correspond to the addition of percent-encoded characters in the URL, described in Section 4.3. The 0 byte increases are an artifact of HTTP2 header compression, described in Section 5.1. A larger increase of approximately 20 bytes occurs after about 12 requests. At

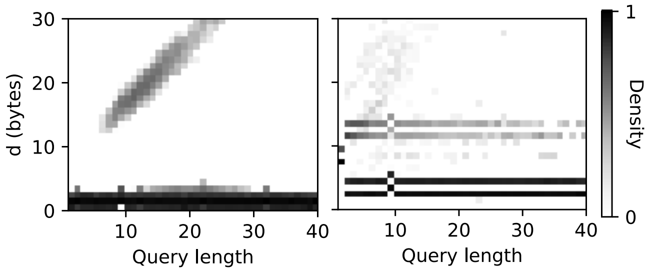

Figure 3: Density of packet size difference between successive autocomplete requests for Google (left) and Baidu (right).

this point, an additional “gs_mss” parameter with the partially completed query is added to the URL. This results in a sudden increase of about 20 bytes: 8 bytes for “&gs_mss=” and 12 bytes for the query. The request then continues to increase by about 1 byte per character thereafter.

The autocomplete packet sizes of Baidu typically increase by either 2 or 4 bytes per character, with a larger increase of 7 or 9 bytes at the beginning of the sequence. After the first request, an additional parameter “pwd=” referring to the previous query is appended to the URL. For example, if the user types “th”, the first request will contain “wd=t” followed by “wd=th&pwd=t”, resulting in a 7 byte increase (6 bytes for “&pwd=t” and 1 byte for “h”). A 4 byte increase corresponds to the addition of escaped characters in the URL, which occupy 3 bytes. Baidu requests also occasionally include a new cookie not present in previous requests, resulting in a larger increase of either 11 or 13 bytes.

Both search engines include a parameter that keeps track of the request number. In Google, this parameter is “cp=”, where “cp” increments with each request (see Figure 1), and Baidu uses the “csor=” parameter. On the 10th request, “cp=9” becomes “cp=10”, resulting in an additional 1 byte increase.

4.2 Keystroke detection

Since autocomplete request size is monotonically increasing, keystroke detection could be performed by finding the longest increasing subsequence (LIS) of packet sizes which has an efficient solution through dynamic programming [44]. However, the LIS fails to capture the fact that packets typically increase by a fixed amount and that two successive packets may be the same size due to HTTP2 header compression. To that end, we generalize the LIS problem to that of finding the longest subsequence accepted by a sequence detector DFA based on observations in the previous section.

We define a DFA that accepts a sequence of packet size differences generated by the autocomplete of each search engine. The DFA for Google autocomplete packets is shown in Figure 4, where edges denote a constraint on d that must be met to traverse to the next state. States a and b correspond to

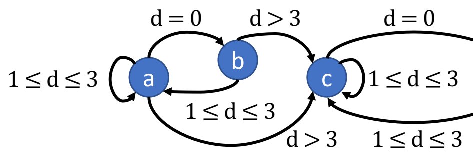

Figure 4: DFA that accepts a sequence of packet size differences generated by autocomplete in Google search.

increases of between 0 and 3 bytes prior to the large increase from the addition of the "gs_mss" parameter, and states *c* and *d* are reached after the large increase. The absence of a recurrent connection on states *b* and *d* indicate that two consecutive non-increases cannot occur. This DFA takes as input a sequence of packet size differences, and if at any point an unreachable state is met, it rejects the sequence.

Let the longest automaton subsequence (LAS) be the longest subsequence accepted by the DFA. Keystrokes are detected by finding the LAS in the sequence of packet sizes. The LAS is determined efficiently through dynamic programming in a similar manner to that of the LIS problem. Let F be an acceptor DFA and L_i the longest subsequence accepted by F ending in the ith packet. Assume the LAS ending in element L_i must necessarily be part of the solution if it contains L_i (optimal substructure). Then L_i need only be computed once and may be considered as the prefix to any other subsequence L_j where $j > i$ (overlapping subproblems). We then need only check if the DFA that accepted the sequence ending in packet i can transition to packet j. Note that in general, these assumptions may not hold and thus the dynamic programming solution might be suboptimal; however, we found this method to work well in practice and leave for future work a formal treatment of the LAS problem.

4.3 Tokenization

Tokenization is the process of delineating words in the sequence of autocomplete requests. Since we assume the search query to be made of English words separated by a Space, this enables the following stages of attack (dictionary pruning, word identification, and beam search) to be conducted at the word level. Like detection, tokenization is a binary classification problem since each packet may be labeled as either a delimiter or part of a word. We consider the Space character as the only delimiter between words.

Percent-encoding is an escape sequence used to represent a character in a URL that is outside the set of allowable characters [8]. A percent-encoded sequence consists of three ASCII characters, "%" followed by two hexadecimal digits. The Space character (ASCII=32) in a URL has percent-encoding "%20". When the user types a Space into the search query field, this escape sequence is appended to the URL causing the uncompressed request packet to increase by 3 bytes.

Google autocomplete packets increase by 2 bytes when the Space key is pressed as a result of HTTP2 header compression. The Huffman code for characters "%", "2", and "0" have bit lengths 6, 5, and 5 respectively, and the sequence "%20" has a total compressed bit length of 16 bits. Tokenization is performed by marking packets that increase by 2 bytes as word boundaries.

Baidu does not use HTTP2, so the escape sequence "%20" occupies 3 bytes. However, since the previous query is included in each request, when a Space is pressed the packet size increases by 4 bytes: 1 for the new character appended to the "pwd" parameter, and 3 for "%20" appended to the "wd" parameter. This also occurs twice in a row since when another letter key is pressed following the Space, "%20" is then appended to the "pwd" parameter. For example, see the URL and sizes of the third and fourth packets in Figure 1 (right), which demonstrate two consecutive 4 byte increases. Tokenization of Baidu queries is achieved by detecting the first of any two consecutive 4 byte increases.

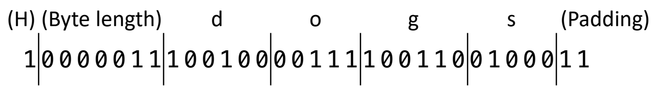

Figure 5: Huffman encoded string literal "dogs" in HPACK.

5 Dictionary pruning

We describe a side channel that leverages the static Huffman code used in HTTP2 header compression. This enables pruning the dictionary, but is only applicable to Google which supports HTTP2. Baidu does not currently support HTTP2.

5.1 Incremental compression side channel

HPACK is the HTTP2 header compression format, which uses a static Huffman code to encode string literals [39]. A Huffman code is a near-optimal lossless compression scheme. Symbols are encoded by bit string with length based on the frequency of the symbol. Huffman codes are prefix-free, such that the code for a symbol is not the prefix to any other. The encoded string becomes the concatenation of all encoded symbols, avoiding the need for symbol delimiters. In HPACK, the encoded string is padded with between 0 and 7 bits to align with the nearest octet boundary.

The static Huffman code in HPACK was determined using a large sample of HTTP headers, and all HPACK implementations must use the same Huffman code defined in the specification [39]. The sizes of lowercase letters range from 5 bits for frequently used characters, such as "e" and "t", to 7 bits for infrequent characters, such as "j" and "z". As an example, the compressed string literal "dogs" is shown in Figure 5. The encoded symbols occupy 6+5+6+5=22 bits, which is then padded with 2 bits for a total size of 3 bytes.

It was previously determined that size alone does not leak a considerable amount of information in HPACK [50]. Let h_i be the bit length of the ith symbol in a string as specified by the static Huffman code and $b = \sum h_i$ the total bit length of the compressed string. The size b reveals only that the string must be some linear combination of encoded symbols to achieve the same compressed size. For example, the string "fish" has compressed length 6+5+5+6=22 bits, exactly the same compressed size as "dogs" in Figure 5.

Less than 0.05 bits per character are revealed in this way, making an HTTP2 compression side channel impractical [50]. This estimate is actually an upper bound since compressed string literals in HPACK are padded to the nearest octet. Instead of b, an adversary observes byte size $B = \frac{p+\sum h_i}{8}$ where $0 \leq p \leq 7$ is an unknown amount of padding to align the compressed bit string with the nearest octet.

However, the query in a sequence of autocomplete requests grows incrementally. Each request contains a single new character appended to the URL path, which then passes through header compression before being sent to the server. We refer to this as *incremental compression*. As a result, instead of total size B, an adversary observes the sequence of cumulative byte sizes $B_1, \ldots, B_n$ of the compressed query after each new character is appended. Due to differences in the size of each symbol, different words grow at different rates and the cumulative byte size sequence can reveal the query.

To leverage the information leaked through incremental compression, we compare the observed sequence of cumulative byte sizes to the cumulative sizes of every word contained in the dictionary. The sizes of words in the dictionary are precomputed for each possible amount of padding, which is unknown to the attacker. Words for which the observed sequence never occurs can be eliminated from the dictionary.

Let d_i be the observed size increase in bytes of the ith request packet and let $B_j = \sum_{i \leq j} d_i$ for $1 \leq j \leq n$ be the observed cumulative size up to the nth request. The sequence $B_1, \ldots, B_n$ characterizes the size growth of a word after undergoing incremental compression. The cumulative byte size sequence $B_1^{w,p_0}, \ldots, B_n^{w,p_0}$ is computed for each word w in the dictionary, given by

$$B_j^{w,p_0} = \left\lfloor \frac{p_0 + \sum_{i \leq j} h_i}{8} \right\rfloor \tag{3}$$

where $0 \leq p_0 \leq 7$ is an unknown amount of padding applied to the compressed URL *prior* to the request containing the first character of the word. The observed size sequence $B_1, \ldots, B_n$ is compared to every sequence $B_1^{w,p_0}, \ldots, B_n^{w,p_0}$ in the dictionary to discover potential matches and eliminate words that could not have been typed by the user.

An example is shown in Figure 6 where the user typed a 4 letter word with cumulative byte size $[1, 2, 3, 3]$. Comparing this sequence to the dictionary, there are two potential matches: the observed sequence $[1, 2, 3, 3]$ appears for the word "dogs" with padding $p_0 = 0$ and for the word "guns" with $p_0 = 0$ or $p_0 = 1$. It's therefore possible that the query contains either the word "dogs" or "guns". However, the user definitely did not search for "cats" since the sequence $[1, 2, 3, 3]$ is not attainable for the word "cats" under any padding p_0. The total size alone does not reveal this much information since all words in the dictionary could have the same total compressed size as the query (3 bytes) given some unknown amount of padding.

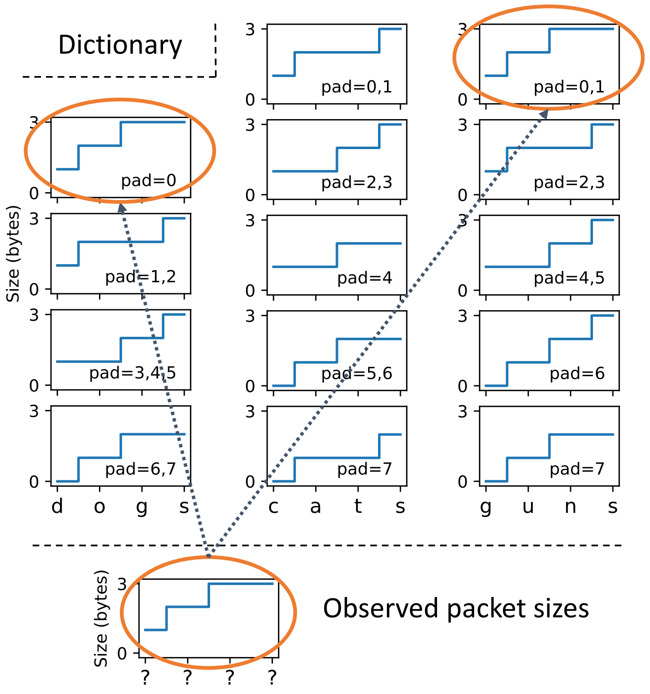

Figure 6: Dictionary pruning. The dictionary contains every possible sequence of cumulative packet size, determined for each word in the dictionary under each unknown prior padding amount (0 to 7 bytes). The cumulative size of an observed query is compared to each sequence in the dictionary. Words that don't have any matches to the observed sequence are eliminated. The observed sequence $[1, 2, 3, 3]$ matches "dogs" with no padding and "guns" with 0 or 1 byte padding; "cats" has no matches and can be safely eliminated.

Note that in general, if $h_i \leq p_{i-1}$, then $d_i = 0$, where p_i is the padding applied after the ith character. That is, when the bit length of a new character is equal to or less than the previous amount of padding used, the packet size will remain the same. Since the lengths of lowercase ASCII characters range from 5 to 7 bits, an increase of at least 1 byte is guaranteed when $p_{i-1} < 5$. It is also never the case that $d_i = 0$ and $d_{i+1} = 0$, i.e., every two consecutive requests must increase by at least one byte.

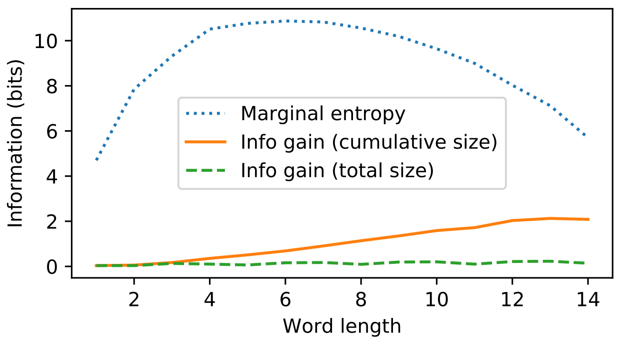

Figure 7: Information gain from an incremental compression side channel, where the cumulative size of a string is exposed, compared to a conventional compression side channel, where only the total size is exposed.

5.2 Pruning and information gain

To measure the impact of this side channel, we determined the expected information gain using a dictionary of 12k common English words and compare this to the information gained from total size alone. Given observed cumulative byte size sequence $\mathbf{B} = B_1, \ldots, B_n$, the probability of each word in the dictionary may be computed by Bayes' formula,

$$P(w|\mathbf{B}) = \frac{P(\mathbf{B}|w)\,P(w)}{P(\mathbf{B})} \tag{4}$$

where $P(\mathbf{B}|w)$ is the probability of sequence $\mathbf{B}$ given word w, $P(w)$ is the marginal probability of word w, and $P(\mathbf{B})$ is the marginal probability the sequence $\mathbf{B}$. Note that multiple byte size sequences could be observed for a particular word depending on the amount of padding used. For example in Figure 6, $P([1,2,3,3]|\text{"guns"}) = \frac{2}{8}$ since the sequence $[1,2,3,3]$ is possible for the word "guns" with paddings of 0 and 1 out of 8 possible padding amounts. In the same example, the marginal $P([1,2,3,3]) = \frac{3}{24}$ since the sequence $[1,2,3,3]$ appears 3 times in the dictionary with 24 precomputed sequences (3 words $\times$ 8 padding amounts). Words for which $P(w|\mathbf{B}) > 0$ are retained in the dictionary in the later stages of the attack and words for which $P(w|\mathbf{B}) = 0$ are eliminated.

From $P(w|\mathbf{B})$, the conditional entropy $H(w^n|\mathbf{B})$ is determined for words of length n. Information gain is given by $I(w^n;\mathbf{B}) = H(w^n) - H(w^n|\mathbf{B})$, where $H(w^n)$ is the marginal entropy of words of length n. We assume each word has an equal probability of occurrence, i.e., $H(w^n)$ has maximum entropy. The information gain is shown in Figure 7. Note that information gain from total byte size B is negligible as previously reported [50]. However, the information gain from cumulative size increases for longer words due to the "uniqueness" of the cumulative byte sizes revealed through incremental compression. These gains lead to more accurate query identification.

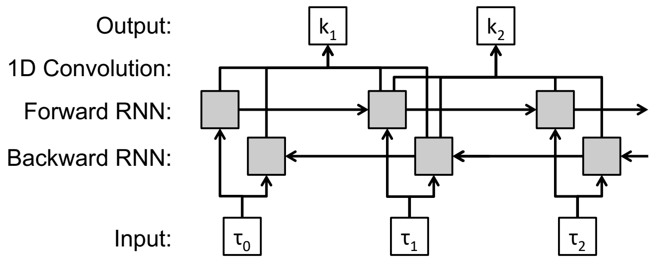

Figure 8: Neural network architecture that predicts n keys from $n+1$ packet inter-arrival times.

6 Word identification and beam search

In the last stages of KREEP, packet inter-arrival timings are used to predict which words the user typed. Word probabilities are determined for the remaining words in the dictionary after pruning, and these probabilities are combined with a language model in a beam search to generate hypothesis queries.

6.1 Word identification from timings

Since each autocomplete request is triggered by a key-press event, packet inter-arrival times faithfully preserve key-press latencies. These latencies are used to predict which keys the user pressed. Unlike previous work which considered either each latency in isolation [47], or words in a limited dictionary [29], we define a model that predicts key probabilities considering their surrounding context and also able to recognize words not seen during training.

We use a three-layer neural network to predict key probabilities. Generally, each word of length n has $n+1$ packet inter-arrival times since a Space precedes the first character and follows the last character. The model takes as input the sequence of latencies τ_i for $0 \le i \le n$ and predicts $P(k_i)$, the probability of each key k_i for $1 \le i \le n$.

The first layer of the network is a bidirectional recurrent neural network (RNN) with gated recurrent units (GRU) that takes as input the sequence of $n+1$ time intervals. The second layer is a 1-dimensional convolutional layer with kernel size 2 and no padding. The convolutional layer reduces the size of the output from $n+1$ to n. The last layer is a dense layer with softmax activation that predicts the probability of each key (26 classes) at each time step. This architecture is shown in Figure 8.

The network architecture was motivated by several factors. The use of a bidirectional RNN ensures that the predictions at key i are made within the context of latencies preceding and following i. The convolutional layer with kernel size 2 combines the latency immediately before and after key i, reducing the size of the sequence from $n+1$ (number of latencies) to n (number of keys). Note that while generally a word of length n has $n+1$ latencies, the first and last words in the query each have n latencies due to missing the leading Space and trailing

Space, respectively. We augment the missing intervals with the mean latency obtained over the entire training dataset.

Word probabilities are determined from the sequence of key probabilities output by the network. The probability of word w is the joint probability of all keys in that word,

$$P(w|\tau) = \prod_{k_i \in w} P(k_i) \tag{5}$$

where τ is the sequence of observed latencies. Making predictions at the key-level and then calculating word probability by the joint key probability has several advantages. First, the number of output classes in the network remains small (26 keys) compared to the number of possible words (over 12k). Second, the probability of any word can be determined whether or not it was contained in the dataset used to train the model. In this way, the dictionary used to generate hypothesis queries is independent of the key identification model.

Finally, learned features may be shared across words. For example, if a particular pattern of latencies is indicative of the sequence "th", the model can learn to recognize "th" in different words such as "the", "there", "beneath", and so on. If instead predictions were made at the word level, these features would have to be learned separately for each word.

6.2 Language model and beam search

In the last stage, word probabilities are combined with a language model to generate hypothesis queries in a beam search.

We assume the query to be a sequence of N words w_i for $1 \leq i \leq N$ and take advantage of the fact that some words are more likely to follow others in natural language. As an example, consider trying to predict an 8-letter word that follows the sequence "recovering from a _". The probability of words such as "sprained" and "fractured" should be relatively higher than other words such as "purchase" and "position".

The use of a language model enables constraints of English language to be leveraged in conjunction with word probabilities from packet timings. A language model estimates the probability of a word given the words that preceded it, denoted by $P(w_i|w_1 \dots w_{i-1})$. We combine the language model with the keystroke timing model to determine the probability of an entire query $\mathbf{w} = [w_1, \dots, w_N]$, given by

$$P(\mathbf{w}) = \prod_{w_i \in \mathbf{w}} P(w_i|\tau) P(w_i|w_1 \dots w_{i-1})^{\alpha} \tag{6}$$

where α is a parameter that controls the weight of the language model. Smaller α places more weight in the packet interarrival timings, while larger α places more weight on the language model. In this work, we found α in the range of 0.2 to 0.5 work well and we use $\alpha = 0.2$. The language model is a 5-gram model with Kneser-Ney smoothing [21] trained on the Billion Word corpus [11].

Determining the sequence with maximum *a posterior* probability (MAP) is NP-hard due to the exponential growth of the search space. It is also unlikely that the MAP sequence itself exactly matches the true query. Instead, KREEP generates a list of hypothesis queries using a beam search. Beam search is a breadth-first greedy search algorithm that maintains a list of top candidates (the "beam") as it progresses the search tree.

For each token, all the words in the dictionary are appended to each hypothesis in the beam, which starts with the empty string. This results in a list of $W \times D$ candidates, where W is the beam width and D is the size of the dictionary. The W sequences with highest likelihood are retained, and the rest discarded. This repeats until the last token is reached, at which point the search returns a list of W hypothesis queries. We use a beam width of 50. To measure the performance of KREEP, we determine the rate at which the query is correctly identified among the 50 hypotheses as well as the minimum edit distance in the list of 50 hypotheses to the true query.

7 Results

In this section, we describe our data collection setup and evaluate attack performance. KREEP is first tested under ideal conditions. We then evaluate performance with increasing levels of simulated network noise and propose a simple padding defense to mitigate attack success.

7.1 Data collection

We built a system that captures network traffic while a query is typed into a search engine with autocomplete. The measurement setup consists of a keystroke dataset previously collected from human subjects, browser automation with Selenium WebDriver, and a process to replay keystrokes by writing keyboard events to `/dev/uinput` in real time.

To train the neural network, we used a subset of a publicly available keystroke dataset collected from over 100k users typing excerpts from the Enron email corpus and English gigaword newswire corpus [15]. From this dataset, we retained 83k users with US English locale on either desktop or laptop keyboards and QWERTY keyboard layout.

To simulate search queries, we randomly selected 4k phrases between 1 and 20 words in length containing only letters and the Space key. This selection contains a wide variety of typing speeds, ranging from 1.5 to 22 keys per second. Of the 4k phrases, 3k are unique. They contain a total of 1717 unique words ranging from 1 to 14 characters with an average word size of 6 characters. None of the users in the evaluation data appeared in the dataset used to train the neural network.

Each capture proceeded as follows. The web browser was opened and cookies cleared before starting the capture process (tshark). One second after the capture began, the website was loaded using Selenium. There was then a two second delay before replaying the keystrokes. The keystroke sequence was replayed by writing the sequence of key events to the

	Google		Baidu	
	Chrome	Firefox	Chrome	Firefox
Detect F-score	99.99	99.96	99.62	99.98
Perfect detect rate	99.72	98.70	96.35	99.52
Token F-score	97.26	95.45	96.85	97.33
Perfect token rate	81.12	74.89	86.70	88.30

Table 1: Keystroke detection and tokenization F-scores (%) and rates (%) of achieving perfect accuracy (F-score=100%).

`uinput` device with delays between each event that correspond to the original keystroke sequence. The data collection was performed on an Ubuntu Linux desktop machine with kernel version 4.15 compiled with the CONFIG_NO_HZ=y option, which omits scheduling clock ticks when the CPU is idle [1]. This ensures keyboard event times are replayed with high fidelity and not quantized due to the presence of a global system timer.

We captured 4k unique queries on search engines Google and Baidu, both of which default to an HTTPS connection and generate autocomplete requests upon key-press events. All results were obtained on the encrypted traffic: TLSv1.3 for Google and TLSv1.2 for Baidu. Both sites leak information through the size of the TLS records, which includes the size of the payload plus a fixed amount for the authentication code (GMAC). Thus, TLS preserves differences in payload length, although TLSv1.3 does contain a provision for record padding to hide length [40].

To understand how the browser itself might affect network timings, the data collect was performed in both Chrome (v.71, with QUIC disabled) and Firefox (v.64). The captured dataset contains a total of 16k queries (4k queries $\times$ 2 search engines $\times$ 2 web browsers), obtained over approximately 7 days. During this time, we did not experience any rate limiting. However, a small number of captures did miss some of the outgoing traffic (< 1%). The unsuccessful captures were repeated until success.

7.2 Attack performance

The first step of the attack is to detect keystrokes. Keystroke detection accuracy is reported separately for each website in each browser in Table 1. In both websites and browsers, keystrokes are detected with near perfect accuracy with a high rate of achieving perfect detection. Tokenization F-scores are also shown in Table 1. The rates of achieving perfect tokenization are strictly lower than that of detection since tokenization is applied after detection.

We examined the cases in which tokenization failed. We found that false positives in Google were due mainly to rollover of the `String Length` field in the HPACK header, which specifies the size in bytes of a compressed string. In HPACK, the string length starts as a 7-bit integer (see Figure 5). When the number of compressed bytes exceeds $2^7 - 1$, an additional byte is allocated for the string length, resulting in an overall increase of 2 bytes (+1 from the String Length increase and +1 from the new character in the query). Since it is generally not known where this rollover occurs, we cannot distinguish whether the 2 byte increase was due to `String Length` rollover or the addition of a percent-encoded Space.

False negatives in both Google and Baidu were due mainly to larger changes in packet size coinciding with a Space. In Google, this occurs when the "gs_mss" parameter is added to the query in the same request as a Space, and in Baidu, from the inclusion of a cookie that was not previously present. These larger changes (> 10 bytes) mask the change in size due to the Space key (2 or 4 bytes).

Following detection and tokenization, the dictionary is pruned, word probabilities from packet inter-arrival timings are determined, and hypothesis phrases are generated in a beam search. Attack success critically depends on accurate keystroke detection and tokenization. This is because the later stages of the attack assume that word lengths have been correctly identified. If the wrong word lengths have been determined, due either to a failure in detection or tokenization, then the correct query cannot be identified.

This behavior is shown qualitatively in Figure 9. In this example, perfect detection and tokenization result in hypothesis queries that have the correct word lengths and low edit distance to the true query. When either a false negative or false positive detection error occurs, the hypothesis queries will have a different length than the true query. In Figure 9 (middle), the 7th packet (containing the 1st "r" in "recovering") is incorrectly labeled as non-keystroke. As a result, the third word in the hypothesis has 9 letters instead of 10. This results in sequences that have relatively high edit distance to the true query. Tokenization errors have a similar effect in that word lengths in the hypothesis will not match the query. In Figure 9 (right) the 11th packet (containing the 2nd "e" in "recovering") is incorrectly labeled as a Space. The hypothesis queries have the same total length as the true query but differ in word lengths, resulting in relatively high edit distance.

The proportion of attacks in which the true query is identified among the hypotheses queries, analogous to a top-50 classification accuracy, is shown Table 2. We also determined the minimum edit distance for each search engine as a function of query length and compare this to a baseline attack in which the timing and language model probabilities are ignored. Baseline performance is obtained by generating 50 ran-

Google		Google (no prune)		Baidu	
Chrome	Firefox	Chrome	Firefox	Chrome	Firefox
15.83	15.13	14.20	13.55	12.85	12.63

Table 2: Top-50 classification accuracy: % of queries that are correctly identified among the 50 hypothesis queries.

Perfect detection/tokenization		Keystroke detection false negative		Tokenization false positive	
he is recovering from a sprained	0	to be president from a position	18	is to learn from such a position	23
he is recovering from a strained	1	to be president from a business	17	is to learn from such a purchase	23
he is recovering from a fracture	7	to be president just a fraction	22	is to learn more from a position	20
he is recovering from a position	7	to be president from a possible	18	is to learn from such a pressing	22
he is recovering from a possible	7	to be president from a southern	18	is to learn from such a practice	21

Figure 9: Query hypotheses in three different scenarios: perfect detection and tokenization (left), false negative keystroke detection (center, the 7th packet is missed), and false positive tokenization (right, the 11th packet is labeled as a Space). The edit distance to the true query "he is recovering from a sprained", is shown to the right of each hypothesis.

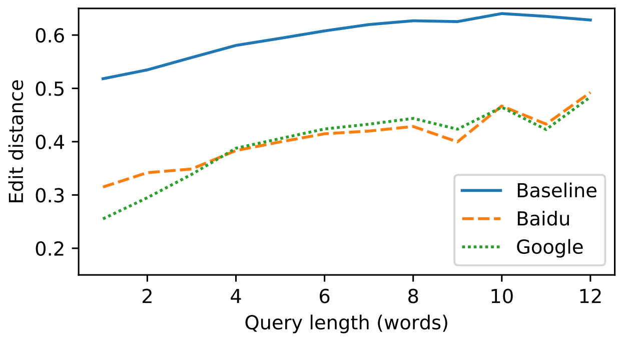

Figure 10: Minimum edit distance (the closest query among 50 hypotheses to the true query) vs query length.

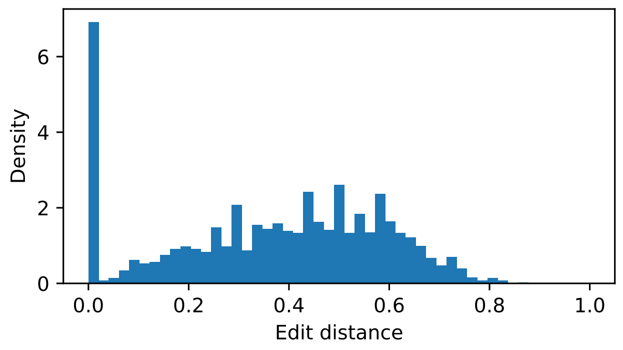

Figure 11: Minimum edit distance distribution. Two modes indicate that KREEP either exactly identifies a query (0 edit distance) or performs near the baseline (0.55 edit distance).

dom hypotheses, choosing dictionary words the same length as the detected tokens. Note that this baseline still uses information gained through keystroke detection and tokenization. These results are shown in Figure 10.

Generally, the difficulty in identifying the query increases with query length. The hypotheses have an average minimum edit distance of 0.37 to the true query. Note that edit distance reduces to Hamming distance for strings of equal length, and perfect detection (F-score of 100%) is achieved in about 98% of queries. Therefore, 0.37 edit distance is roughly a 63% key identification accuracy. We did not find any significant difference in performance across browsers, but did achieve overall higher query identification rates on Google due information leaked through incremental compression.

We found the example in Figure 9 to be representative of attack success which generally had polarized outcomes: the hypotheses were either very similar to or very different from the true query. This behavior is revealed in the distribution of minimum edit distances shown in Figure 11, which has two modes: one occurring near the baseline (0.55, achieved by guessing random words) and the other at 0.

7.3 Information sources

To better understand the relative contribution of each component, we evaluate attack performance ignoring the packet timings, language model probabilities, or header compression. Considering only queries in Google (Baidu does not support HTTP2), performance is evaluated for three scenarios: using only the packet timings (TM only), using timings and the language model (TM+LM), and using both with dictionary pruning applied (TM+LM+Pruning).

These results are shown in Figure 12 with baseline performance as described in the previous section. The largest gains are achieved with the use of packet timings and language model. The neural network alone identifies words with 19.1% accuracy. Incremental gains are then achieved when the dictionary is pruned.

7.4 Effects of network noise

We tested the robustness of the attack to network noise. Since key identification uses packet inter-arrival times, packet delay variation (PDV) can potentially reduce attack success. PDV corresponds to changes in network latency, which can obfuscate the key-press timings in packet inter-arrival times. In this regard, variations in routing delay potentially provide a natural defense to remote keystroke timing attack.

The Laplace distribution has previously been proposed as a model for PDV [60]. We simulate PDV by drawing samples from a Laplace distribution parameterized by the mean absolute deviation (MAD). The simulated PDV is added to

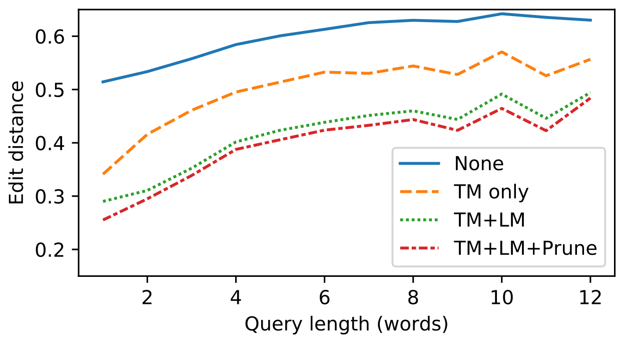

Figure 12: Performance with/without the use of the timing model (TM), language model (LM) and dictionary pruning.

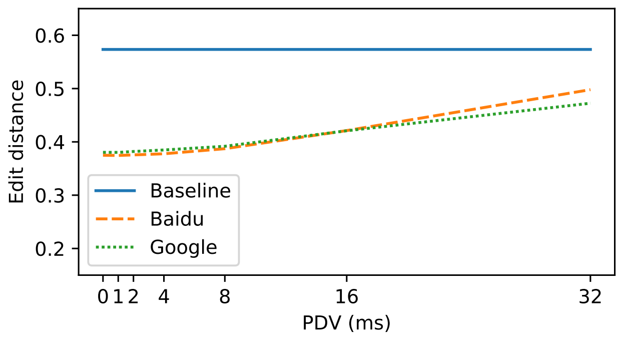

Figure 13: Effects of packet delay variation. Baseline ignores packet timing, uses only packet size to generate hypotheses.

the captured packet times before attempting to identify the query with KREEP. Performance as a function of increasing PDV is shown in Figure 13. The attack is relatively robust to PDV less then 8 ms, but approaches baseline performance with PDV in excess of 32 ms.

7.5 Effects of padding

With the attack being robust to low levels of network noise, we explored other means of mitigating attack success. Query identification critically depends on accurate detection and tokenization, and chances of attack success can be greatly reduced with a simple padding scheme.

We simulate random padding by modifying the captured packet sizes. The size of each autocomplete packet is increased by 1 byte with probability 0.5. The sizes of other

Detect F-score		Token F-score		Min edit distance	
Original	Padded	Original	Padded	Original	Padded
99.89	94.46	96.72	51.23	37.76	61.32

Table 3: Effects of randomly padding packets with 0 or 1 byte.

packets in the trace remain unchanged such that the padding defense could be implemented entirely in the client side autocomplete logic.

The effects of this defense nearly double the minimum edit distance, shown in Table 3. While this scheme does not greatly reduce the ability to detect keystrokes, it makes tokenization difficult which poisons later stages of the attack. Note that tokenization could also be made more difficult by encoding the Space key as a single character, such as "+" instead of the 3 byte sequence "%20". Search engines Yandex and DuckDuckGo both use this strategy. However, this does not exclude the possibility of tokenization through other means such as timings, an item we leave for future work.

8 Discussion

Search engines with autocomplete are part of a larger class of applications in which the manifestation of human-computer interactions in network traffic can lead to a remote side channel attack. This includes VoIP: as utterances are compressed and transmitted in real time, spoken phrases can be identified in encrypted network traffic [55, 56]; SSH: single characters are transmitted to and echoed back by the server, exposing the timing of key presses [47]; HTTP: unencrypted network traces contain a user's web browsing activity [36, 57]; and HTTPS: in dynamic web applications, server response size can reveal interactions with specific elements on a web page [12].

8.1 Related work

Keystroke timing attacks Keystroke timing attacks were introduced in [47], which considered the identification of key pairs (bigrams) from key-press latencies to aid in password inference. Such an attack is generally possible because of the non-zero mutual information between keys and keystroke timings, e.g., keys far apart are usually pressed in quicker succession than keys that are close together [43]. This behavior generalizes across subjects, similar to other phenomena in human-computer interaction (HCI) such as Fitts' Law [17]. There has been some debate whether a remote keystroke timing attack poses a credible threat [3, 22]. Evidence suggests that while information gain is generally possible, attack success is user-dependent with some users being more vulnerable than others [33, 34].

In [47], a hidden Markov model and generalization of the Viterbi algorithm were used to generate candidate passwords from timings. The key-press latencies used to train the model were recorded in isolation, wherein subjects pressed a key pair as opposed to typing a full password. In addition, the keystrokes were recorded on the host under the assumption that the key-press latencies would be faithfully preserved in the network traffic. Our work confirms that assumption by using timings obtained from actual network traffic and users typing complete phrases instead of isolated bigrams.

There have been numerous works focused on the *detection* of keyboard events (which enables a timing attack), such as through spikes in CPU load [45], cache and memory usage [41], and the `proc` filesystem [23]. Few works have considered remote keylogging attacks [12,58]. In [51], the authors examine the extent to which autocomplete exposes key-press latencies in network traffic and found that multiple observations were required to recover the true latency. In a recent work, we characterized the autocomplete network traffic of five major search engines and measured the correlation between key-press latencies on the host and packet inter-arrival times observed remotely [35], finding search engines Google and Baidu to leak the most information. The findings in [35] partly motivated the development of KREEP.

Since the work [47], several studies have examined timing attacks on password [6, 59] and PIN [29, 30] input. We depart from prior work, which has focused on sequences with maximum prior entropy, by targeting natural language input, which is more susceptible to keystroke timing attack due to a relatively lower prior entropy (roughly 1 bit per char, as noted in [47]). We introduced a method to combine language model probabilities with information leaked through keystroke timings, inspired by the use of language models in conjunction with acoustic models in automatic speech recognition [20]. In addition, our attack combines multiple independent sources of information leakage beyond keystroke timings, including URL escape sequences and HTTP2 header compression.

Compression side channels A compression side channel leverages information leaked through the compression of a plaintext prior to encryption [28]. Because different strings compress to different sizes, compressed size can reveal information about the plaintext. HTTPS exposes the length of an encrypted payload, making it vulnerable to attack when the payload is compressed. There have been several attacks on HTTPS based on this principle.

The CRIME attack exploits compression in TLS and in the now deprecated SPDY protocol [42]. This attack requires a man-in-the-middle vantage in which an attacker inserts a guess for a secret, e.g., an HTTP cookie or a CSRF token, into a message and observes the compressed size. The DEFLATE compression algorithm in SPDY uses redundancy to compress a string [14] such that the compressed size of a packet containing the correct guess will be smaller than an incorrect guess. The BREACH attack leveraged a similar principle for server responses, targeting compression at the HTTP level (e.g., gzip) [18], and the TIME attack used server response time as a proxy to measure response size [9].

HEIST lowered the bar for attack, enabling CRIME-like attacks to be deployed remotely within a victim's web browser [52]. The size of a compressed server response is determined at the application level by examining whether the response time spans multiple round trips, an indication that the entire response exceeded the TCP congestion window. This general technique can be used in a variety of side channel attacks beside guessing secrets, such as determining whether a user is logged into a particular site [52].

DEFLATE, the compression algorithm used in gzip, uses a combination of LZ77 and Huffman coding [14]. To date, all compression attacks against HTTPS have exploited the LZ77 component of DEFLATE, which builds a dictionary from the redundant parts of a string. The Huffman code in DEFLATE has been treated as noise, typically dealt with by making guesses in pairs. For example, to find out whether a secret starts with "p", an attacker guesses "secret=p_" and "secret=_p": if the sizes are the same, then only Huffman coding is used and the guess is wrong; otherwise, if the sizes are different, the LZ77 component was invoked based on redundancy between the first guess and the secret, and only Huffman coding was invoked in the second guess.

HPACK, the header compression format in HTTP2, was designed to be resistant to CRIME-like attacks targeting LZ77 compression, although HTTP2 borrowed many concepts from SPDY [39]. Commonly used header fields are compressed with a dictionary lookup, and string literals are compressed using a static Huffman code, which was previously determined to leak relatively little information [50]. But unlike previous attacks, KREEP leverages the static Huffman code in HPACK rather than an LZ77 dictionary. We found considerably more information is leaked due to several contributing factors:

1. *HTTPS exposes payload size*. HTTPS was previously shown to leak information by exposing the length of an encrypted payload. The HTTPS Bicycle attack uses the size differences between HTTP requests to infer the size of an unknown secret [53]. An attacker simply subtracts the size of all known parts of the request, leaving only the size of the secret. Our attack relies on a similar principle, taking the difference in size between successive autocomplete requests.

2. *Characters are independently compressed*. The size difference between two compressed payloads that differ only by the insertion of a single character reveals the compressed size of that character. However, Huffman encoded strings in HPACK are padded to the nearest octet, mitigating the amount of information that would otherwise be leaked without padding. Since byte, and not bit, size differences are observed, the symbol size is known only to within a margin of error that depends on an unknown amount of padding.

3) *The Huffman code is standard*. Every HPACK implementation uses the same Huffman code, which is publicly available [39]. An attacker needs only to map dictionary words to their cumulative compressed sizes, taking into account the unknown amount of padding applied beforehand. Potential matches to a secret are revealed by comparing its cumulative compressed size to every word in the dictionary.

Search query identification Previous work on identifying search queries has utilized features obtained primarily through traffic analysis. In [37], keywords in search queries are identified over Tor using both inbound and outbound autocomplete traffic. Keystrokes were replayed in a data collection setup similar to ours described in Section 7.1. Packet inter-arrival times were not considered since the replayed keystrokes used random, and not human, timings. Instead, each search query is characterized by packet counts and sizes, inbound and outbound Tor cell counts, and other features specific to Tor traffic. The work of [37] did not attempt keystroke detection but instead focused on the identification of queries that contain a particular keyword from a set of target keywords. With this approach, a query containing any one of 300 target keywords could be identified with 85% accuracy, and individual keywords with 48% accuracy.

We instead aim to reconstruct an entire query rather than identify the presence of some target words, and we leverage information leaked through packet size, which is obfuscated by cell size in Tor traffic. While keystroke detection may be possible in traffic over Tor, for example by detecting traffic that has "keystroke-like" packet inter-arrival times, tokenization and dictionary pruning cannot be applied since the autocomplete packet sizes are masked behind Tor cell sizes. An attack that uses *only* packet inter-arrival times might be feasible in Tor, but would require a different approach than our attack.

While previous work has shown HTTP response size to leak a considerable amount of information about a user's query when autocomplete suggestions are provided [12], we chose to focus only on HTTP requests. In [12], an attacker guesses a victim's query one letter at a time by trying all combinations and matching the server response size. This assumes the attacker can submit queries that induce the same suggestions as the victim received. In practice, this is difficult because autocomplete suggestions depend on the victim's search history and location, among other factors [2]. To our knowledge, KREEP is the first attack targeting autocomplete traffic from the client independent of these factors, relying only on packet inter-arrival times and packet size differences.

8.2 Countermeasures

Keylogging attacks require successful keystroke detection *and* key identification. Therefore, it is sufficient to prevent keystroke detection *or* key identification to counter the attack. We consider the tradeoffs of several countermeasures and how they affect each source of information leakage.

Padding Padding could be applied in two different ways: pad each request by a random amount, or pad to ensure all requests are the same size. To increase keystroke detection false negatives, the pad amounts must be sufficiently large to disguise autocomplete traffic with other background traffic, the size of which is generally not known a priori. Therefore, padding may not be effective to mitigate keystroke detection and does not provide any protection against a timing attack. However, we have confirmed that padding by a small random amount (1 byte with probability 0.5) does effectively mitigate tokenization and incremental compression. Note that padding in this way should be applied only to alphabetic characters and not to the addition of a Space; otherwise, some packets with a Space will increase by 3 bytes (2 bytes + 1 padding byte), while all other packets increase by no more than 2 bytes. The pad amounts should be chosen such that the *observed* packet size differences closely follow a uniform distribution.

Dummy traffic While padding aims to increase detection false negatives, generating dummy traffic aims to increase false positives. A false positive occurs when background traffic is labeled as a keystroke. Generating dummy autocomplete requests with approximately the same size as the actual request would make keystroke detection a difficult task. With each autocomplete request, the client could send a burst of several packets with similar size (within several bytes), randomly ordering the actual request within the dummy request. While an attacker might still be able to perform detection with a low false negative rate, this comes at a cost of an increased false positive rate. This method mitigates tokenization, compression, and timing attacks. The background traffic would overwhelm the actual requests, similar to the generation of dummy keyboard events in KeyDrown [45]. This approach has the cost of increased bandwidth, a tradeoff reminiscent of the anonymity trilemma [13], and requires some cooperation from the server to ignore the dummy requests.

Merge requests Most search engines make an autocomplete request immediately following each new character appended to the input field [35]. Instead, combining multiple characters into a single request would mitigate our attack in two different ways. First, with multiple characters merged into a single request, the number of false negative detection errors must increase since a packet contains multiple keystrokes. This conceals the timing information of all but the last character in the merged request, reducing information leakage through a keystroke timing attack.

Additionally, merged requests effectively eliminate the incremental compression side channel since the increase in packet size corresponds not to a single character but to multiple characters. The compressed size of the merged characters must be some linear combination of symbols in the Huffman code, and as string length increases, the number of combinations grows exponentially [50].

Combining requests could be achieved in several ways: 1) update the list of autocomplete suggestions after every other, or every *n*th, key (similar to Nagle's algorithm, except at the application level); 2) use a polling model with polling rate slower than the user's typing speed (Bing performs polling with 100ms interval, making this attack impractical for fast typists); or 3) trigger callbacks on `keyup` events instead of `keydown` events (DuckDuckGo does this), which merges requests when consecutive keystrokes overlap [35], a typing phenomenon referred to as *rollover* [15]. The drawback in all cases is that merging requests could adversely affect usability since the suggested queries are delayed to the user.

8.3 Limitations and future work

We point out several limitations of our attack, emphasizing the conditions under which it succeeds, and identify ways in which KREEP could be extended or improved.

Other websites In this work, KREEP has only been tested on search engines Google and Baidu. Keystroke detection and tokenization are both application-specific, based on the packet size pattern each search engine emits. Extensions to other search engines or websites would require modification to these components. For websites that aren't vulnerable to tokenization, delimiters might be identified based on packet inter-arrival times (e.g., larger intervals indicate Space, smaller intervals indicate letters).

Other modalities Since autocomplete requests are induced by keyboard events, KREEP is applicable only up to the point when a user stops typing or selects a suggested query. We assumed that no deletions or corrections were made and that the user did not press any non-printable keys, e.g., arrow keys, that cause the caret to change position. However, selecting a query from the provided suggestions does not preclude the possibility of other attacks that incorporate the timing of both autocomplete requests and server responses. It may be the case that the way the user interacts with the autocomplete suggestions also leaks course-grained information, such as user identity or the type of query (navigational, informational, or transactional) [10]. One might also consider the timing of mouse clicks that induce network traffic as a source of information leakage by leveraging a general model that governs click behavior, such as Fitts' Law [17].

Targeted attacks Finally, while we made an effort to evaluate our attack on phrases that are representative of natural language, the content of actual search queries is quite different and varies between users as evidenced by the AOL search dataset [7, 38]. Some strings that have a low probability of occurrence in natural language, such as "www", tend to occur frequently in search queries. This affects the prior probability of each symbol, which must be properly accounted for in the language model. We verified this difference by comparing the frequency of characters in the AOL search dataset to the keystroke dataset we used to evaluate KREEP (which itself borrowed phrases from the Enron email corpus [15]). These are shown in Figure 14. Notably, the frequencies of "w" and "c" in search queries are about twice that of natural language, likely due to the presence of navigational queries to a specific URL, such as "www.example.com". Likewise, Space characters in search are about half as frequent compared to natural language. In a targeted attack, the language model in KREEP could be tailored towards a particular victim, leveraging information such as the victim's native language, geographic location, and public blog entries.

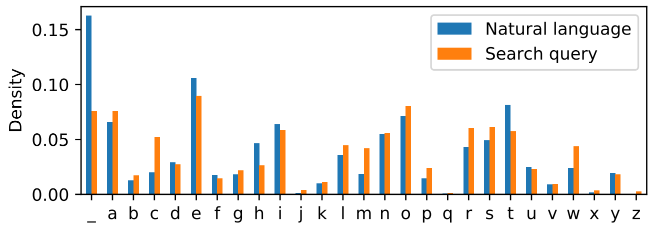

Figure 14: Character frequency in natural language (Enron corpus) compared to search queries (AOL search dataset).

9 Conclusion

KREEP leverages multiple independent sources of leaked information to identify search queries in encrypted network traffic. Autocomplete request packets are detected based on packet size; queries are tokenized by detecting the presence of URL-escaped characters; keys are identified based on packet inter-arrival times; and impossible words are eliminated from a dictionary based on incremental compression. Despite many moving pieces, the attack obtains a reasonable success rate, recovering more than half the characters in a query on average. But more importantly, the pieces that contribute to this attack present some starting points for future research.

The static Huffman code used in HTTP2 header compression leaks more information than previously thought [50] when incremental changes are made to a string in the header. This kind of attack is not limited to search engines with autocomplete but could apply to any website with dynamic content that updates incrementally. It will be beneficial to identify other web applications that exhibit incremental compression. Besides websites that provide search suggestions, this could include mapping services, which modify the geographic coordinates in a URL as the user drags the map center location, or websites that autosave the contents of a text field.

Likewise, websites that generate network traffic in response to user input events may be vulnerable to timing attack. Sites that support remote document editing, such as Google Docs, frequently transmit the document state from the client to the server. When this process is event driven, i.e., triggered by `keydown` events, the network traffic can leak information about the user's actions or document content. Similarly, chat applications that aim to provide real-time updates about a conversation partner's activity, e.g., by displaying a notification that "X is typing", also risk exposing keystroke timings in network traffic if those notifications are directly driven by the conversation partner's keystrokes.

Availability

KREEP is available at https://github.com/vmonaco/kreep. The keystroke dataset is publicly available [15].

Acknowledgements

We thank the anonymous reviewers and our shepherd for valuable feedback during the review process. The manuscript was much improved based on insightful discussions with colleagues Justin Rohrer and Robert Beverly, who also provided comments on an early draft.

References

[1] NO_HZ: Reducing Scheduling-Clock Ticks. http://web.archive.org/web/20190208124417/https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt. Accessed: 2019-02-08.

[2] Search using autocomplete. http://web.archive.org/web/20190209193857/https://support.google.com/websearch/answer/106230?hl=en. Accessed: 2019-02-09.

[3] Timing analysis is not a real-life threat to ssh secure shell users. http://web.archive.org/web/20010831024537/http://www.ssh.com/products/ssh/timing_analysis.cfm. Accessed: 2019-02-09.

[4] Eytan Adar. User 4xxxxx9: Anonymizing query logs. In *Proc of Query Log Analysis Workshop, International Conference on World Wide Web*, 2007.

[5] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *Proc. IEEE Symp. on Security & Privacy (SP)*, pages 3–11. IEEE, 2004.

[6] Kiran S. Balagani, Mauro Conti, Paolo Gasti, Martin Georgiev, Tristan Gurtler, Daniele Lain, Charissa Miller, Kendall Molas, Nikita Samarin, Eugen Saraci, Gene Tsudik, and Lynn Wu. SILK-TV: Secret information leakage from keystroke timing videos. In *Computer Security*, pages 263–280. Springer International Publishing, 2018.

[7] Michael Barbaro, Tom Zeller, and Saul Hansell. A face is exposed for aol searcher no. 4417749. *New York Times*, 9(2008):8, 2006.

[8] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. Technical report, jan 2005.

[9] Tal Be'ery and Amichai Shulman. A perfect crime? only time will tell. *Black Hat Europe*, 2013, 2013.

[10] Andrei Broder. A taxonomy of web search. In *ACM Sigir forum*, volume 36, pages 3–10. ACM, 2002.

[11] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[12] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proc. IEEE Symp. on Security & Privacy (SP)*, pages 191–206. IEEE, 2010.

[13] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latencychoose two. In *Proc. IEEE Symp. on Security & Privacy (SP)*. IEEE, 2018.

[14] P. Deutsch. DEFLATE compressed data format specification version 1.3. Technical report, may 1996.

[15] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. Observations on typing from 136 million keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI 18*. ACM Press, 2018.

[16] Tobias Fiebig, Janis Danisevskis, and Marta Piekarska. A metric for the evaluation and comparison of keylogger performance. In *Proc. 7th Usenix Conf. on Cyber Security Experimentation and Test*, pages 7–7. USENIX Association, 2014.

[17] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology*, 47(6):381, 1954.

[18] Yoel Gluck, Neal Harris, and Angelo Prado. Breach: reviving the crime attack. 2013.

[19] David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English gigaword. *Linguistic Data Consortium, Philadelphia*, 4(1):34, 2003.

[20] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International conference on machine learning*, pages 1764–1772, 2014.

[21] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 690–696, Sofia, Bulgaria, August 2013.

[22] Michael Augustus Hogye, Christopher Thaddeus Hughes, Joshua Michael Sarfaty, and Joseph David Wolf. Analysis of the feasibility of keystroke timing attacks over ssh connections. *Research Project at University of Virginia*, 2001.

[23] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Proc. IEEE Symp. on Security & Privacy (SP)*, pages 143–157. IEEE, 2012.

[24] Bernard J. Jansen, Amanda Spink, and Tefko Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing & Management*, 36(2):207–227, mar 2000.

[25] Rosie Jones, Ravi Kumar, Bo Pang, Andrew Tomkins, Andrew Tomkins, and Andrew Tomkins. I know what you did last summer: query logs and user privacy. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 909–914. ACM, 2007.

[26] Sepandar D Kamvar et al. Anticipated query generation and processing in a search engine, 2004.

[27] Stavroula Karapapa and Maurizio Borghi. Search engine liability for autocomplete suggestions: personality, privacy and the power of the algorithm. *International Journal of Law and Information Technology*, 23(3):261–289, jul 2015.

[28] John Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption*, pages 263–276. Springer Berlin Heidelberg, 2002.

[29] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *Proc. 22nd European Symp. on Research in Computer Security*, 2017.

[30] Ximing Liu, Yingjiu Li, Robert H. Deng, Shujun Li, and Bing Chang. When human cognitive modeling meets PINs: User-independent inter-keystroke timing attacks. *Computers & Security*, sep 2018.

[31] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp) iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proc. 18th ACM Conf. on Computer and Communications Security (CCS)*, pages 551–562. ACM, 2011.

[32] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, et al. Quantitative analysis of culture using millions of digitized books. *science*, 331(6014):176–182, 2011.

[33] John V Monaco. Poster: The side channel menagerie. In *Proc. IEEE Symp. on Security & Privacy (SP)*. IEEE, 2018.

[34] John V Monaco. Sok: Keylogging side channels. In *Proc. IEEE Symp. on Security & Privacy (SP)*. IEEE, 2018.

[35] John V Monaco. Feasibility of a keystroke timing attackon search engines with autocomplete. In *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2019.

[36] Christopher Neasbitt, Roberto Perdisci, Kang Li, and Terry Nelms. ClickMiner. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. ACM Press, 2014.

[37] Se Eun Oh, Shuai Li, and Nicholas Hopper. Fingerprinting keywords in search queries over tor. *Proceedings on Privacy Enhancing Technologies*, 2017(4):251–270, oct 2017.

[38] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale*, volume 152, page 1, 2006.

[39] R. Peon and H. Ruellan. HPACK: Header compression for HTTP/2. Technical report, may 2015.

[40] E. Rescorla. The transport layer security (tls) protocol version 1.3. Technical report, aug 2018.

[41] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. 16th ACM Conf. on Computer and Communications Security (CCS)*, pages 199–212. ACM, 2009.

[42] Juliano Rizzo and Thai Duong. The crime attack. In *Ekoparty Security Conference*, 2012.

[43] Timothy A Salthouse. Perceptual, cognitive, and motoric aspects of transcription typing. *Psychological bulletin*, 99(3):303, 1986.

[44] C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[45] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating keystroke timing side-channel attacks. In *Proc. Network and Distributed System Security Symp (NDSS)*, 2018.

[46] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *Proc. 21st Intl. Conf. on Financial*

Cryptography and Data Security (FC), page 11. IFCA, 2017.

[47] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proc. Usenix Security Symp.*, 2001.

[48] Statcounter. Search engine market share china. http://web.archive.org/web/20190209193125/ http://gs.statcounter.com/search-engine-market-share/all/china. Accessed: 2019-02-09.

[49] Statcounter. Search engine market share world-wide. http://web.archive.org/web/20190209193145/ http://gs.statcounter.com/search-engine-market-share. Accessed: 2019-02-09.

[50] Jiaqi Tan and Jayvardhan Nahata. Petal: Preset encoding table information leakage. Technical report, 2013.

[51] Chee Meng Tey, Payas Gupta, Debin Gao, and Yan Zhang. Keystroke timing analysis of on-the-fly web apps. In *Proc. Intl. Conf. on Applied Cryptography and Network Security*, pages 405–413. Springer, 2013.

[52] Mathy Vanhoef and Tom Van Goethem. Heist: Http encrypted information can be stolen through tcp-windows. *Black Hat USA 2016*, page 1, 2016.

[53] Guido Vranken. Https bicycle attack. Technical report, dec 2015. Accessed: 2019-05-10.

[54] He Wang, Ted Tsung-Te Lai, and Romit Roy Choudhury. Mole: Motion leaks through smartwatch sensors. In *Proc. 21st Annual Intl. Conf. on Mobile Computing and Networking (MobiCom)*, pages 155–166. ACM, 2015.

[55] Andrew M. White, Austin R. Matthews, Kevin Z. Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on fon-iks. In *2011 IEEE Symposium on Security and Privacy*. IEEE, may 2011.

[56] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, may 2008.

[57] Guowu Xie, Marios Iliofotou, Thomas Karagiannis, Michalis Faloutsos, and Yaohui Jin. Resurf: Reconstructing web-surfing activity from network traffic. In *IFIP Networking Conference, 2013*, pages 1–9. IEEE, 2013.

[58] Ge Zhang and Simone Fischer-Hübner. Timing attacks on pin input in voip networks (short paper). In *Proc. Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 75–84. Springer, 2011.

[59] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. *analysis*, 20:23, 2009.

[60] Li Zheng, Liren Zhang, and Dong Xu. Characteristics of network delay and delay jitter and its effect on voice over IP (VoIP). In *ICC 2001. IEEE International Conference on Communications. Conference Record (Cat. No.01CH37240)*. IEEE.