Keystroke Timing Analysis of on-the-fly Web Apps

Chee Meng Tey[1], Payas Gupta[1], Debin Gao[1], and Yan Zhang[2]

[1] Singapore Management University
{cmtey.2008,payas.gupta.2008,dbgao}@smu.edu.sg
[2] State Key Laboratory Of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
zhangyan@iie.ac.cn

Abstract. The Google Suggestions service used in Google Search is one example of an interactivity rich Javascript application. In this paper, we analyse the timing side channel of Google Suggestions by reverse engineering the communication model from obfuscated Javascript code. We consider an attacker who attempts to infer the typing pattern of a victim. From our experiments involving 11 participants, we found that for each keypair with at least 20 samples, the mean of the inter-keystroke timing can be determined with an error of less than 20%.

1 Introduction

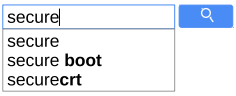


Fig. 1. Suggestions for 'secure'

Rich and complex Javascript (JS) applications provide sophisticated GUI updates and fast client-server communications that approaches the capabilities of traditional desktop applications. For example, Google Instant [1] and Google Suggestion(GS) [2] allow users to view results and suggestions on–the–fly while typing search queries. The front-end JS communicates using HTTP(s) with the back-end server in response to various events such as keypress. Figure 1 and Table 1 shows respectively the GS interface and HTTP requests when a user types in the search term 'secure'. In this paper we explore whether the improved GUI creates a timing side channel. We hope a detailed study of one application yields insights on the threats that may apply to the entire class of such applications.

Related work. Similar side channels attack had been demonstrated by Chen et al. [3] (inferring encrypted JS traffic from packet size) and Song et al [4] (reducing search space of SSH passwords from packet timing). This paper differs from prior work in the following ways. It is the first to analyse JS timing side channels and use it to derive typing patterns, which raise a privacy concern as prior research showed that typing patterns are unique and allows user identification [5,6,7,8]. Moreover, personalized typing patterns improves the SSH attacks

M. Jacobson et al. (Eds.): ACNS 2013, LNCS 7954, pp. 405–413, 2013.

Table 1. Query scenarios: (a) Slow typing. (b) Typing correction. (c) Typing s, e, c, then choosing *secure* from the suggestions. (d) Fast typing (not handled in this paper)

(a)

```
GET /s?...&q=s&
GET /s?...&q=se&
GET /s?...&q=sec&
GET /s?...&q=secu&
GET /s?...&q=secur&
GET /s?...&q=secure&
```

(b)

```
GET /s?...&q=s&
GET /s?...&q=se&
GET /s?...&q=sev&
GET /s?...&q=se&
GET /s?...&q=sec&
GET /s?...&q=secu&
GET /s?...&q=secur&
GET /s?...&q=secure&
```

(c)

```
GET /s?...&q=s&
GET /s?...&q=se&
GET /s?...&q=sec&
     long pause
GET /s?...&q=secure&
```

(d)

```
GET /s?...&q=sec&
GET /s?...&q=secur&
GET /s?...&q=secure&
```

by Song et al. [4] and allows imitation attacks [9] on keystroke biometrics systems [10,7,5,11,12]. JS timing side channels are challenging to analyse because keystroke and network traffic timing are only loosely correlated. This is because JS applications (a) are far slower as compared to native binary applications and (b) typically run in a single threaded co-operative multitasking execution model.

Key results. In the following sections, we study GS's communication model and derive a set of techniques to construct a keypair timing model (probability distribution of keypair intervals) for each pair of keystroke from unencrypted GS traffic. We conducted a user study on 11 participants to collect their keystrokes and timings. Results show that if at least 20 samples of each keypair are available, the recovered mean timing differs from the actual mean by at most 20%. However, the recovered standard deviation is less accurate: with at least 40 samples, the maximum difference is 46%. The accuracy improves with the increase in the pool of samples indicating the effectiveness of long term attacks.

2 Communication Model

Our approach to study the GS communication model is based on both black-box testing and white-box analysis. We used the setup of Figure 2. The client under testing connects to the Google servers in the back-end through a proxy server. For blackbox testing, we captured Google query packets using a packet sniffer [13] installed on the client. For whitebox testing, we hosted a copy[1] of the Google HTML and JS files on another web server and selectively redirect the proxy server [14] to fetch our copy rather than from the actual Google server. This allows us to make arbitrary changes to the scripts for our testing.

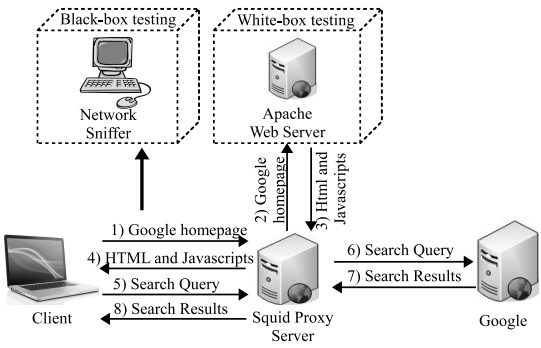


Fig. 2. Setup for black-box and white-box testing

[1] Retrieved Apr 2012.

2.1 Approach

Black-box analysis allowed the quick identification of traffic patterns and content. For example, we quickly found that different network traffic patterns are possible even for the same query (see Table 1). On the other hand, when we analysed the timings of the keypress and the packets, we encountered significant difficulty correlating them. For example, after the user pressed a key, the corresponding HTTP request can be observed on the network from between approximately 3 ms to over 100 ms later with 2 distinctive frequency peaks at around 7 ms and 45 ms. Whitebox analysis is therefore necessary.

Our approach is to first manipulate the obfuscated source code using the tool JSBeautifier [15]. The decision to host a separate copy of the script files in Figure 2 allowed us to make arbitrary changes to the JS source code independently of the Google servers. Next, we use the console logging feature of Firebug to pinpoint the code that initiated the HTTP requests. This formed the starting point for subsequent investigations, where we incrementally assign meaningful symbols to the variables and functions through (a) monitored calls to standard functions, and (b) selectively breaking execution and examining the call stack and variables. Please note that our investigation focuses specifically on the timing aspects. Hence we did not deobfuscate all the script code involved in GS. The rest of this section documents our findings.

2.2 Communication Model Obtained

JS uses an event driven execution model [16]. For GS, there are 3 classes of event handlers of interest. $\underline{\texttt{H}_{\texttt{poll}}}$ is a handler for polling events. The polling is setup and removed when the query input box receives and loses focus respectively. Although the specified polling interval is 10 ms, the actual firing interval fluctuates. The reason is likely to be due to other events firing and executing, thereby delaying the execution of this handler. $\underline{\texttt{H}_{\texttt{ui}}}$ handles UI events, e.g., `keydown`, `keypress`, `keyup`, etc. The same handler code fires for different events but with different closure scope. The handler function for the `keydown` event, named $\texttt{H}^{\texttt{kd}}_{\texttt{ui}}$, is installed during GS code initialization. The GS code includes a mechanism to defer execution of a function. $\underline{\texttt{H}_{\texttt{defer}}}$ handles the events which are deferred. The 2 key parts to this are the `postMessage` JS function and an array of deferred functions ($\texttt{Arr}_{\texttt{defer}}$). At load time, GS setups a `message` event handler. This handler fires when a message is posted to it using `postMessage`. When fired, it removes the first function from $\texttt{Arr}_{\texttt{defer}}$ and executes it. If $\texttt{Arr}_{\texttt{defer}}$ is not empty, it posts a message to itself and exits. Any JS code deferring execution calls a wrapper function `defer` which first pushes the function to defer onto the bottom of $\texttt{Arr}_{\texttt{defer}}$ and then post a message to $\texttt{H}_{\texttt{defer}}$.

When a user presses and releases a key, JS fires four events in this order: `keydown`, `keypress`, `input`, `keyup`. Under normal circumstances, $\texttt{H}^{\texttt{kd}}_{\texttt{ui}}$ uses the deferred execution mechanism to invokes a function named $\texttt{H}_{\texttt{x}}$ to send out the network traffic. However, it is also possible for $\texttt{H}_{\texttt{poll}}$ to run before $\texttt{H}^{\texttt{kd}}_{\texttt{ui}}$. In such a case, $\texttt{H}_{\texttt{poll}}$ invokes $\texttt{H}_{\texttt{x}}$ directly (without deferring execution) to send out the query.

Regardless of the path taken, H_x is executed at most once for each keystroke. The end result is a race (to execute H_x) between the synchronous mechanism of H_{poll} and the asynchronous mechanism of H_{ui}^{kd}, resulting in the introduction of a variable delay. The race is won mostly by H_{ui}^{kd}. Another factor affecting the execution delay is the number of task on each execution path. For example, the first keypress for GS also updates the UI in preparation for not just the suggestions of GS, but also the results of Google Instant. This additional code increases the execution delay by approximately 6 times (∼45 ms).

Listing 1.

```
trace Hx
    if xhr_timer not pending then
        xhr_m enter
            ...
        exit
    end if
end trace

trace TimerEvent
    Call xhr_m
end trace

procedure xhr_m
    if unsent_query then
        xhr_i enter
            ...
            send query to Google
            ...
        exit
        timeout ← compute_timeout
        create TimerEvent
    end if
end procedure
```

A third factor affecting the delay is submission throttling [17]. Most search engines used this technology to limit the amount of search traffic to their website while the user is typing the query. In the case of GS, regardless of whether H_{poll} or H_{ui}^{kd} won the race, H_x is always invoked. The role of H_x is to send queries and receive results from the Google servers. Listing 1 shows how submission throttling is implemented in GS when H_x is invoked. The sending mechanism of GS uses a timer named xhr_{timer}. This timer is initially cleared. When H_x is invoked it checks this timer. If xhr_{timer} is cleared, H_x calls a sub function xhr_m to send out the query immediately. Otherwise, it exits without sending any HTTP traffic. When xhr_m runs, it sets up the timer xhr_{timer} to call itself (xhr_m) again after a timeout value. The detailed computation of the timeout is out of the scope of this paper, but on a fast network, this value is 100 ms. After this timer is set, xhr_m will not run again until the timer expires. Any keystrokes typed during this time accumulate and are sent together in the same HTTP request when the timer expires. If xhr_m runs but does not find any unsent query (that is, between the previous and current invocation of xhr_m, the user did not press any key), it does not set any new timer. xhr_{timer} therefore becomes cleared again. If a new key is now pressed, xhr_m will again send it out without delay. The described process then repeats itself. The implication is that correlation between keystroke and packet timing is poor whenever xhr_m is in timer mode.

3 Recovery of Keypair Timing Model

Section 2 identifies the timeout mechanism and atypical execution path as major noise contributors. Packet timings thus affected are considered unreliable. Figure 3 shows that if we discard the unreliable timing, the delay between pressing of keystroke and sending of packet becomes significantly more consistent. (t_{ks} and t_{pkt} refer to the keystroke and packet timing respectively.) This allows the recovery of the derived keypair timing model.

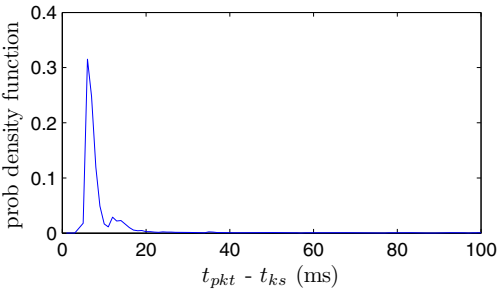


Fig. 3. Noise model

For each keypair, the recovery process involves (a) identifying the corresponding packet-pairs, (b) determining if each packet timing is reliable, (c) choosing packet-pairs where the earlier timing is reliable, (d) further dividing the chosen packet-pairs into a set where the latter packet timing is reliable and another set where it is not, (e) computing the mean and variance of the packet-pair timing model and finally (f) applying a correction to the variance to obtain the derived keypair timing model. This process requires an assumption of normally distributed timing models which are independent. Prior work [4] investigating keypair timing model found the normal distribution to be a reasonable approximation. In step (d), the size of the first set (reliable latter packet timing) is denoted by N_o. The size of the complementary set is denoted by N_u.

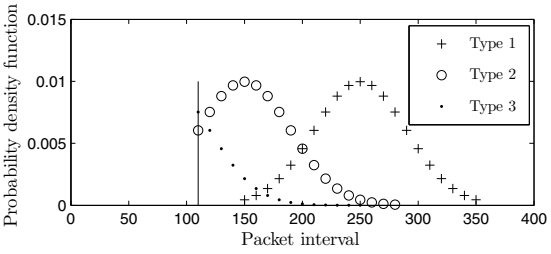


Fig. 4. Different scenarios for the building of $\texttt{DTM}_{\texttt{key}}$ from $\texttt{TM}_{\texttt{pkt}}$

In step (e), depending on the values of N_o and N_u, there can be 4 different scenarios, 3 of which are shown in figure 4. $\texttt{TM}_{\texttt{key}}$ denotes the keypair timing model, $\texttt{TM}_{\texttt{pkt}}$ denotes the packet-pair timing model, and $\texttt{DTM}_{\texttt{key}}$ denotes the derived keypair timing model, which is an approximation of $\texttt{TM}_{\texttt{key}}$ obtained by applying a variance correction to $\texttt{TM}_{\texttt{pkt}}$. In type 1 scenarios, $N_u = 0$. The mean and variance are calculated directly from the observed intervals.

For type 2, $N_u < N_o$. The timeout translates to a cutoff time beyond which certain intervals are not observed. The peak though is still visible. We first estimate the median from N_u and the observable parts of the distribution. We next obtain the mean which is equal to the median. We estimated the missing part of the distribution by reflecting the observable part about the mean. From the reconstructed distribution, we can calculate the variance.

For type 3, $N_u \geq N_o$. The peak is not visible. Our aim is to fit a normal distribution (with unknown mean $\bar{x}$ and std. deviation s) based on the interval observations on the right tail. Let l denote the cutoff time (timeout + a small allowance). $\bar{x}$, s, l, α are related by $l = \bar{x} + \alpha s$, where α is a multiplier s.t. for a standard normal distributed random variable X, $\Pr(X > \alpha) = N_o/(N_o + N_u)$. The curve fitting iterates over a possible list of values for $\bar{x}$, and computes the corresponding s. The values providing the best fit (least squares) is chosen.

In type 4 scenarios, $N_o = 0$. The mean of the keypair timing is far less than the timeout, resulting in no interval observations. Hence it is not plotted in Figure 4. To recover the timing for a keypair such as $c_1 - c_2$ where c_i denotes a key pressed, we need to have the parameters of another 2 distributions: the keypair $c_2 - c_3$ and the triplet $c_1 - c_2 - c_3$. The latter 2 distributions would be observable if there exists c_3 such that $c_2 - c_3$ is much longer than the timeout.

Given two independent normally distributed random variables X and Y, the random variable $Z = X + Y$ is also normal [18] with mean $\bar{z} = \bar{x} + \bar{y}$ and standard deviation $s_z = \sqrt{s_x^2 + s_y^2}$. The relation between $c_1 - c_2$, $c_2 - c_3$ and $c_1 - c_2 - c_3$ is analogous to that of X, Y and Z. Therefore, if we let Z and Y represent the distribution for $c_1 - c_2 - c_3$ and $c_2 - c_3$ respectively, we can obtain the mean and standard deviation of the unobservable $c_1 - c_2$ from X.

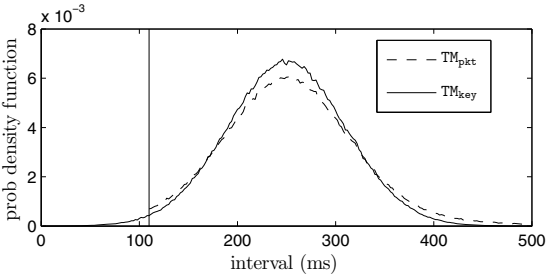


Fig. 5. Difference in the *p.d.f.* of $\texttt{TM}_{\texttt{key}}$ compared to the corresponding *p.d.f.* of $\texttt{TM}_{\texttt{pkt}}$

In step (f) we need to apply a correction to the variance but not the mean. This is because there is residual noise even after accounting for the timeout and atypical execution path. This noise affects both timing observations of a packet-pair. It cancels out for the interval mean, but adds to the variance. Figure 5 shows this effect for a keypair. To compute the required variance correction, we use a simple heuristic. A Monte Carlo simulation based on the model of Section 2 computes the observed variance for a set of variances. The differences are stored in a table and looked up whenever a correction is needed.

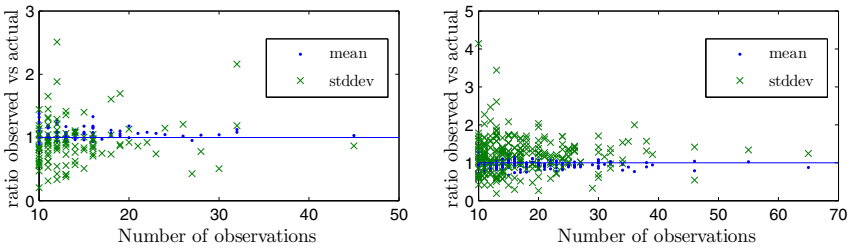
4 User Study

To verify the theory of Section 3, we conducted a user study. 11 participants are asked to install a plugin on their browser which captures the keystroke timings of GS queries. The duration of the study ranges from 32 to 49 days. Users are allowed to inspect and delete any sensitive entries in the capture log before submission. Towards the end of the study, users with too few queries were given a chance to go through a Q&A worksheet using Google to find the answers. This is so that they get more opportunities in using Google search. The collected keystrokes are anonymised and post processed to retain only English alphabets and the SPACE char. Queries with BACKSPACE are broken up. The resulting logs are consolidated on a single machine running Ubuntu 11.10 (AMD Athlon(tm) 64 X2 Dual Core Processor 4000+ 2110 MHz with 3 GB RAM). The keystrokes in each query are injected programmatically using the `uinput` [19] interface and the corresponding query packets are collected.

Table 2. Statistics of user study. Q: total number of queries by user. KS: total keystrokes typed. KP: total keypairs typed. TP: total 3 char sequence. KP_{obs}: sum of N_o for all keypairs. TP_{obs}: sum of N_o for all triplets. N_{sig}: total number of keypair/triplets for which $N_o \geq 10$. This is also the number of recovered $p.d.f.$

S/N	Q	KS	KP	TP	KP_{obs}	TP_{obs}	N_{sig}
1	502	3114	2612	2110	642	487	6
2	421	2666	2245	1824	682	506	7
3	1206	6607	5401	4195	2447	1715	93
4	688	4243	3555	2867	601	403	6
5	593	3604	3011	2418	1368	993	34
6	774	4592	3818	3044	1752	1284	58
7	405	2517	2112	1707	733	561	8
8	696	4610	3914	3218	1181	893	29
9	327	2163	1836	1509	270	185	1
10	1041	6042	5001	3960	2359	1697	96
11	700	3964	3264	2564	1233	853	29

The outcome of the user study is shown in Table 2. There is a positive correlation between the number of queries submitted and the number of $p.d.f.$ (last column) recovered in $\texttt{DTM}_{\texttt{key}}$ for each participant. This suggests that long term collection of queries would recover far more $p.d.f.$ than our user study. The outcome of the methods for type 1 to type 2 are shown in Figure 6. Relatively fewer samples were collected for type 3 and type 4 due to the low probability of finding observations at the tail and finding both triplet and keypair accounts. The outcome for these methods are omitted due to brevity of space. Generally, the mean can be recovered accurately although larger observations tend to result in more accuracy. The variance, on the other hand, is less accurate, particularly for fewer observations. Like the mean, however, the accuracy improves as the observations increases.



(a) Type 1 recovery (b) Type 2 recovery

Fig. 6. Recovery of $p.d.f.$ from various types of packet observations

4.1 The Optimal Timeout

Given that the current timeout value of 100ms allows the derivation of $\texttt{DTM}_{\texttt{key}}$ (from $\texttt{TM}_{\texttt{pkt}}$), we also investigated the possible countermeasures. These countermeasures are equally applicable to any JS application with rich interactivity that wishes to deny potential adversary the opportunity for UI events harvesting. We conducted a Monte Carlo simulation using the set of keystroke data collected from the user study. We varied the GS timeout and computed the simulated packet timing based on the noise model and the findings of Section 2.

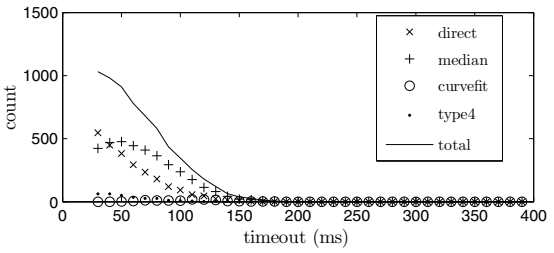


Fig. 7. Variation of the total recovered keypair or triplets for all users given a particular timeout setting

Figure 7 shows the variation of the count of recovered keypairs and triplets vs the timeout. Choosing a timeout figure of 200-250 ms eliminates most observations, but the responsiveness is more than halved. Given that in Listing 1, $\texttt{xhr}_{\texttt{m}}$ exits timeout mode whenever there is no keystroke activity in the previous timeout cycle, an alternative is to increase the number of timeout cycles to 3 while keeping the timeout unchanged at 100 ms. This eliminates all observable intervals without affecting the responsiveness.

5 Limitations

In our study, the keypair timing model is based on keydown-keydown intervals. Many biometric authentication techniques use such intervals [10,11,5]. Our work therefore affects such systems. However, biometric authentication is not limited to just keydown-keydown metrics. Keydown-keyup, keyup-keydown and even keypress pressure are examples of alternatives. For the first 2, active attacks injecting malicious Javascript code can capture both keydown and keyup, but this is not investigated in this paper. Keypress pressure however, cannot be measured by Javascript applications and are therefore unaffected.

The keystroke injection part of the user study was done on a dedicated machine. We therefore did not model the additional execution delay that may result if the machine is also running multiple compute intensive processes concurrently.

The description of GS [2] indicated that it may behave differently in different geographical locations. We did not manage to isolate any geographically specific code during our investigations. This is either due to our limitations or different source code was delivered to different locations. Our findings therefore apply only to geographical locations with similar settings as our evaluation environment.

6 Conclusions

In this paper, we investigated the recovery of personalized keystroke timing information using GS. We found that it is possible to construct a user's typing pattern from the timing of the queries sent over the network. This is of concern because the availability of typing pattern is a prerequisite for (a) achieving the best outcome in timing side channel attacks and (b) imitation attacks on keystroke biometrics. It can also be used to identify users. This suggests that logs recording network traffic of interactive Javascript applications should be considered confidential and handled accordingly. Otherwise, the operators of search

engines as well as proxy server administrators can mine the typing pattern of their users from the traffic logs. We suggest that designers consider alternative options such as multiple timeout cycles to shut down the leak effectively.

References

1. Google Instant, http://goo.gl/WI9Zu
2. Autocomplete, http://goo.gl/jv3fQ
3. 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. IEEE Computer Society, Washington, DC (2010)
4. Song, D.X., Wagner, D., Tian, X.: Timing analysis of keystrokes and timing attacks on ssh. In: Proceedings of the 10th conference on USENIX Security Symposium, SSYM 2001, vol. 10, p. 25. USENIX Association, Berkeley (2001)
5. Araujo, L., Sucupira, J. L., Lizarraga, M., Ling, L., Yabu-Uti, J.: User authentication through typing biometrics features. Trans. Sig. Proc. 53(2), 851–855 (2005)
6. Killourhy, K.S.: A Scientific Understanding of Keystroke Dynamics. Dissertation, Carnegie Mellon University (2012)
7. Peacock, A., Ke, X., Wilkerson, M.: Typing patterns: A key to user identification. IEEE Security and Privacy 2(5), 40–47 (2004)
8. Monrose, F., Rubin, A.D.: Keystroke dynamics as a biometric for authentication. Future Gener. Comput. Syst. 16(4), 351–359 (2000)
9. Tey, C.M., Gupta, P., Gao, D.: I can be You: Questioning the use of Keystroke Dynamics as Biometrics. In: Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA (February 2013)
10. Joyce, R., Gupta, G.: Identity authentication based on keystroke latencies. Commun. ACM 33(2), 168–176 (1990)
11. Haider, S., Abbas, A., Zaidi, A.: A multi-technique approach for user identification through keystroke dynamics. In: IEEE International Conference on Systems, Man and Cybernetics, SMC 2000, pp. 1336–1341 (2000)
12. Killourhy, K., Maxion, R.: Why did my detector do *that*?!: predicting keystroke-dynamics error rates. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 256–276. Springer, Heidelberg (2010)
13. Tcpdump, http://www.tcpdump.org
14. Squid, http://www.squid-cache.org/Intro/
15. JSBeautifier, http://jsbeautifier.org/
16. DOM Events, http://en.wikipedia.org/wiki/DOM_events
17. Mahemoff, M.: Ajax Design Patterns. O'Reilly Media, Inc. (2006)
18. Normal Sum Distribution, http://goo.gl/wfaMz
19. Uinput, http://thiemonge.org/getting-started-with-uinput