Behav Res (2017) 49:1686–1695
DOI 10.3758/s13428-017-0958-7

Metronome LKM: An open source virtual keyboard driver to measure experiment software latencies

Pablo Garaizar[1] · **Miguel A. Vadillo**[2]

Published online: 15 September 2017

Abstract Experiment software is often used to measure reaction times gathered with keyboards or other input devices. In previous studies, the accuracy and precision of time stamps has been assessed through several means: (a) generating accurate square wave signals from an external device connected to the parallel port of the computer running the experiment software, (b) triggering the typematic repeat feature of some keyboards to get an evenly separated series of keypress events, or (c) using a solenoid handled by a microcontroller to press the input device (keyboard, mouse button, touch screen) that will be used in the experimental setup. Despite the advantages of these approaches in some contexts, none of them can isolate the measurement error caused by the experiment software itself. Metronome LKM provides a virtual keyboard to assess an experiment's software. Using this open source driver, researchers can generate keypress events using high-resolution timers and compare the time stamps collected by the experiment software with those gathered by Metronome LKM (with nanosecond resolution). Our software is highly configurable (in terms of keys pressed, intervals, SysRq activation) and runs on 2.6–4.8 Linux kernels.

Keywords Virtual keyboard · Experimental software · Open source software

✉ Pablo Garaizar
garaizar@deusto.es

[1] Faculty of Engineering, University of Deusto, Avda. Universidades 24, 48007 Bilbao, Spain

[2] Universidad Autónoma de Madrid, Madrid, Spain

Over the last decades, computers have become an essential tool to conduct psychological experiments. In these experimental setups, participants typically interact with several input devices connected to a computer running an operating system that enables using the experiment software. As can be seen, several layers separate participants' interactions from the experiment software controlling those interactions. Researchers must bear in mind each of these abstraction levels when designing experimental paradigms with high timing requirements, as all of them can be the origin of measurement error.

The quality of the measures returned by the experiment software is assessed in benchmarking studies, in which the performance of the software is tested under well-known conditions and its measurements are matched against the expected results. For the specific case of reaction-time measures, benchmarking studies require a source of user events that can be controlled experimentally.

Several methods can be used to generate user interaction events under well-controlled experimental conditions. None of them is free from limitations, but they also have interesting advantages that differentiate them from each other and make them particularly suitable for specific situations. In the following sections, we provide a brief overview of extant methods to generate user input and we explain why generating keyboard events through a virtual keyboard driver can be an adequate solution to test experiment software under a wide range of conditions.

Experimental participants

An obvious means to gather realistic user interaction data is to recruit participants and measure their performance under controlled conditions. This is a common approach in behavioral sciences. For instance, Reimers and Stewart

(2007) used this method to analyze the accuracy and precision of reaction time data collected with a Flash program in a sample of undergraduate students who participated in exchange for course credit. Similarly, Keller, Gunasekharan, Mayo, and Corley (2009) recruited voluntary participants to test the accuracy of their Java-based WebExp package for the design of online experiments.

The main advantage of this approach is its external validity. The results can be easily generalized to real studies because the data are gathered under natural conditions. However, an important shortcoming is that participants' responses might not be reliable, due to practice effects or to the use of unrepresentative samples. In many cases, it is impossible to calibrate properly the experimental devices or to assess the impact of (systematic or unsystematic) errors arising from the hardware. Because of this, we think that this strategy must be considered complementary to approaches based on lower levels.

Specialized hardware

In light of these limitations, some researchers have resorted to specialized hardware to simulate user interaction, which grants more control over the generated user data. The Black Box Toolkit (Plant, Hammond, & Turner, 2004) is an excellent example. This hardware can send signals through the parallel port (Schneider, Eschman, & Zuccolotto, 2002) or, alternatively, activate a button located in an input device that is connected to the computer running the experimental software (Plant & Turner, 2009). Other researchers have connected photo-sensors to "artificial respondents" (Häusler, Sommer, & Chroust, 2007; Rorden & Hanayik, 2014; Schubert, D'Ausilio, & Canto, 2013) or solenoids that generate keyboard presses (Neath, Earle, Hallett, & Surprenant, 2011) to simulate an automated experimental subject capable of interacting in a consistent and reproducible manner with the application or technology under study.

The most interesting feature of this alternative approach, as compared to using real participants, is that it allows researchers to replicate the study as many times as necessary, manipulating any variable that could be relevant and assessing its impact on the final result.

Although this is possibly the best strategy for benchmarking the whole experimental setup, it is not free from problems, either. The signal generated by an external device might not replicate exactly the interaction of real users. Potentially, sources of bias in the input device employed by users might go unnoticed in the system that sends and collects signals from the external generator (e.g., sending keyboard events through the parallel port to test an experimental setup in which USB keyboards will be used). The opposite situation is also possible—that is, sources of bias might affect how signals are sent or received from the generator without producing a similar effect on the input devices employed by users (e.g., using an Arduino-based device to generate key events and send them through the USB port to test an experimental setup in which a response pad connected to a PIO12 card will be used). Consequently, the measurements collected with this method might not always converge with those of the previous method. This problem is avoided when the external generator can be directly connected to the input device itself (e.g., hacking a USB keyboard to connect a specific key to the external generator), but this is not always possible.

Off-the-shelf hardware

Given that the use of specialized hardware is expensive, both in terms of time and money, some researchers have used general-purpose hardware, readily available in most equipment, to generate user-interaction data. For instance, Eichstaedt (2001), among others, assessed the temporal precision of Java applets using the typematic repeat rate of a conventional keyboard. Typematic repeat allows researchers to generate a large number of keypresses per second automatically. It can be activated easily by keeping a key pressed for a time longer than the typematic delay. Once this interval is surpassed, key-presses will be generated automatically according to the typematic repeat rate, until the key is released or a different key is pressed.

Although this might look like a simple and valid method to generate constant user interaction data without sophisticated equipment and without a complex setup, this approach is unsatisfactory for several reasons. Firstly, typematic repeat is not implemented consistently in different keyboards. In the case of AT-PS/2 keyboards, it is defined in the communication protocol (Chapweske, 2003), but in USB keyboards it is implemented by the operating system, using the packet send rate and the number of received packets to estimate the duration of keypresses. Therefore, typematic repeat is not always generated by the external device itself, but by the operating system. Secondly, general-purpose keyboards are usually manufactured with low quality standards. In contrast to other input devices like joysticks or clickers, keyboards are expected to require a low response rate (below 500 presses per minute) and, hence, manufacturers have little motivation to build them with optimal components. Finally, the popularization of wireless keyboards adds another layer of complexity, given that the electromagnetic field is more exposed to interference than a regular cable connection. Because of all these reasons, we think that the typematic repeat rate should not be used to assess the accuracy and precision of an application.

Specialized software

Bearing in mind the shortcomings of all the approaches based on hardware, several researchers have developed software alternatives for the automatic generation of user interaction data. Together with simple systems for the generation of keypresses or mouse events, such as uinput or Keypresser, there are complete suites for the automatization of user input specifically designed for the Web, such as Selenium (Badle et al., 2012).

The fact that the generation of user interaction does not depend on hardware allows this approach to circumvent all the biases and latencies inherent to hardware. However, the main limitation of this strategy is that the software used to generate user interaction data can influence the performance of the application under study, and vice versa. Ideally, this kind of mechanisms should be located at a software-architecture level that avoids any influence from other software components (e.g., at the kernel level) and must rely on timing mechanisms that remain immune to overload at the user application level.

Metronome LKM

To overcome the limitations of the approaches presented in the previous section, we decided to develop a keyboard driver relying on high-resolution timers that can generate configurable keyboard events. The reasons for this choice are that: (1) none of the conventional input devices (e.g., keyboard, mouse, touch screen) can work with submillisecond time delays (Bhalla & Bhalla, 2010; Crosbie, 1990; Damian, 2010; Plant, Hammond, & Whitehouse, 2003; Plant & Turner, 2009; Segalowitz & Graves, 1990), which imposes a lower limit on the precision of time measurements (Rorden & Hanayik, 2014); (2) the use of uncommon communication ports, like the parallel port, requires adapting the application to this input device, which is not always possible; (3) the fact that this driver runs at the kernel level avoids the impact of overload at the user application level; (4) this approach allowed us to use timers (i.e., high-resolution timers) and timing functions (i.e., `ktime`) with precision and accuracy below the millisecond; and (5) from the perspective of software, the experimental program will receive the automatically generated keyboard events as if they had been generated by a real user.

We chose GNU/Linux as a development platform due to the large volume of publicly available information for the development of drivers for the Linux kernel (Bovet & Cesati, 2005), the simplicity of the C code needed to develop a loadable kernel module (LKM), and the highly flexible means offered to deploy and test several types of virtual machines, kernel debuggers, and registry systems. GNU/Linux is also a useful platform for behavioral researchers, given that some of the most popular software packages for the deployment of psychological experiments, such as PsychoPy (Peirce, 2007, 2009), OpenSesame (Mathôt, Schreij, & Theeuwes, 2012), or Psychophysics Toolbox (Brainard, 1997; Pelli, 1997), are explicitly designed to run on this operating system. JavaScript libraries for Internet-based psychological research (e.g., jsPsych, lab.js; de Leeuw, 2015; Henninger, Mertens, Shevchenko, & Hilbig, 2017) can also run under GNU/Linux. In general, GNU/Linux and open source software are becoming increasingly popular among scientists from many fields, because they reduce substantially the costs of research (Pearce, 2012).

Metronome LKM is published under an open source license (GPLv3), which grants developers and researchers permission to use, adapt, audit or correct it free of charge. The code repository is publicly available at https://github.com/txipi/MetronomeLKM.

Configuration parameters

Metronome LKM is a driver designed for the kernel of Linux 2.6.21 (or higher) that relies on high-resolution timers to generate accurate and precise keyboard events. The parameters of the driver can be defined when it is loaded or through the sys virtual file system (i.e., `/sys/module/metronome/parameters/`) and it can be enabled or disabled using a system request with the SysRq key. In addition to generating the required keyboard events, the driver also logs each of them with ns-resolution timing functions (i.e., `ktime`) in the kernel log file.

Several parameters of Metronome LKM can be defined during run time. See Table 1 for a comprehensive list. Once the driver is loaded, it can be activated with the combination of keys previously defined (SysRq + D by default) or by means of the command `echo 1 > /sys/module/metronome/parameters/metronome_status`. Metronome LKM can be stopped by following the same procedure: either pressing the predefined combination of keys (SysRq + D) or through the command `echo 0 > /sys/module/metronome/parameters/metronome_status`.

How Metronome LKM works

When Metronome LKM is loaded into the kernel of the operating system, function `metronome_init` requests the necessary system resources to register a new input device with `input_allocate_device` and `input_register_device`. If the registration is successful, Metronome LKM sets the system request that enables or disables the generation of keyboard events and configures the monotonic high-resolution timer (with the `hrtimer_init` function and the kernel's `CLOCK_MONOTONIC` policy, which takes into account time drifts in multiprocessor systems).

Table 1 Configuration parameters for Metronome LKM

Name	Description
metronome_delay	Delay (in nanoseconds) between key input events. Several delays can be defined for different key events (metronome_delay, metronome_delay_2, metronome_delay_3). Initially set to 1E09 (i.e., 10^9 ns = 1s).
metronome_key	Key event that will be generated by Metronome LKM. Several key events can be defined (metronome_key, metronome_key_2, metronome_key_3). Initially set to KEY_SPACE (i.e., space bar).
metronome_sysrq	Combination of keys (in addition to SysRq) that triggers the system request to enable or disable the generation of keyboard events. Initially set to "d," which means that the request is triggered by the combination SysRq + D.
metronome_status	Status (active or inactive) of the key event generation process. Initially set to 0 (inactive).

Once this process is complete, `metronome_hrt_callback` confirms that there has been no delay in the management of the timer (using the `hrtimer_forward` function), generates the requested keyboard events (using `input_report_key` and `input_sync`), logs the time when they were submitted (with `ktime_get`) and resets the high-resolution timer (returning the value `HRTIMER_RESTART`). If the user enters the combination of keys defined to change the status of Metronome LKM at any time, this will trigger `sysrq_handle_metronome`, which toggles between active and inactive status.

How to use Metronome LKM

Metronome LKM is a Loadable Kernel Module for Linux-based systems. Therefore, a Linux distribution (e.g., Debian, Ubuntu, or Gentoo) with a 2.6.21 (or higher) kernel is a prerequisite to use it. In addition, we recommend installing several software packages before downloading, compiling and installing Metronome LKM. In this section, we explain briefly the steps that should be followed in a Debian- or Ubuntu-based Linux distribution, but these guidelines can be adapted easily to other distributions.

To install the required packages, the user must have root access permission. All Linux systems allow the user to get root access using the `su` (i.e., super user) command. Some recent systems use the `sudo` (i.e., super-user do) command as the default method to perform administration tasks. In Ubuntu, for example, any command preceded by the word `sudo` will be executed with root access permission. Therefore, to install the software packages required by Metronome LKM, the user should run `sudo apt-get install git build-essential linux-headers-generic dkms`. Once these packages have been installed, Metronome LKM can be downloaded from the GitHub repository running `git clone https://github.com/txipi/metronome.git -depth 1`. To compile and install it, the user must access the metronome directory (using `cd metronome`) and launch the compilation and installation process with `sudo make install`. Once the process is complete , Metronome LKM can be installed into the operating system kernel with the command `insmod metronome.ko`.

As we explained above, Metronome LKM enables and disables the automatic generation of keyboard events using the special key SysRq (in many keyboards, this function is shared with the PrintScreen key and is activated with the key combination Alt + PrintScreen). However, many Linux distributions prevent the use of the SysRq key on the default setup, as it grants permission to perform administration tasks like resetting the system. In these cases, SysRq should be activated with the command `sudo sysctl -w kernel.sysrq=1`. Alternatively, this change can be done permanently creating a file called `/etc./sysctl.d/90-sysrq.conf` with the instruction `kernel.sysrq = 1`.

Finally, if the user wants to keep record of many keyboard events generated by Metronome LKM, the kernel ring buffer where this information is stored (which can be checked with `dmesg`) might not be large enough. To expand the size of the buffer, the user must add the `log_buf_len` modifier to the system startup manager. In Ubuntu, this can be done changing the file `/etc./default/grub` in the specific line where the variable `GRUB_CMDLINE_LINUX_DEFAULT` is defined, setting it to "`quiet splash log_buf_len =16M`", where `16M` refers to the desired buffer size.

Study 1

The goal of the present study is to test the validity of Metronome LKM to generate keyboard events collected by a simple X-Window-System (i.e., the Linux Graphical User Interface) application. In Study 2, we compare those measurements with the results obtained by a JavaScript Web application based on standard DOM events time stamps running on Google Chrome and Mozilla Firefox.

Method

Apparatus and materials We installed Metronome LKM on an Ubuntu Linux vanilla distribution, unconnected to the Internet and isolated from external sources of asynchronous events. On this setting, we run the Logkeys X-Window, an

application for the X-Window-System graphical interface explicitly developed to log keyboard events. This is achieved through the functions `XGrabKeyboard` (to get key events on a X-Window-System environment) and `gettimeofday` (to get time stamps with microsecond accuracy).

Procedure The specific steps we followed were (1) we loaded Metronome LKM with the appropriate parameters (in the default configuration, the driver remains inactive initially), (2) launched the application under study, (3) activated the driver through a system request (SysRq), (4) logged the input events with the application under study and with the keyboard driver itself, and (5) analyzed the data. The same procedure was followed with different keypress intervals (i.e., one keypress every 1,000, 500, 100, 50, 10, 5, and 1 ms) until 10,000 or more samples had been collected.

Results and discussion

The results of the tests conducted with Metronome LKM and Logkeys X-Window are shown in Table 2, where all values refer to milliseconds. As can be seen, the mean errors obtained with Metronome LKM are well below a millisecond, around nanoseconds in the worst cases, with standard deviations of 4.499 μs in the worst case. The mean errors obtained with Logkeys X-Window are also below a millisecond (between 4.599 and 0.178 μs), but the standard deviations are noticeably larger than those obtained at the kernel level (between 4.214 μs and 0.112 ms). These values can be used as a baseline for the assessment of performance under Web applications running on Google Chrome 17 and Mozilla Firefox 10 in Study 2.

Study 2

The goal of Study 2 was to benchmark a JavaScript Web application based on standard DOM events time stamps running on Google Chrome 17 and Mozilla Firefox 10 using Metronome LKM.

Method

Apparatus, materials, and procedure As in Study 1, we installed Metronome LKM on an Ubuntu Linux vanilla distribution unconnected to the Internet and isolated from external sources of asynchronous events. On this setting, we run a JavaScript Web application that relies on the time stamps of standard DOM Events (Pixley, 2000) to log the time elapsed from the previous time stamp to the moment when the keyboard event is registered (i.e., the keypress event). The procedure was otherwise identical to Study 1.

Results and discussion

Table 3 summarizes the results obtained with Google Chrome 17. Although the mean errors obtained by Metronome LKM are generally close to a nanosecond, their standard deviations are systematically larger than the errors observed in Study 1. This pattern is also observed for the tests conducted on Mozilla Firefox 10, which suggests that the small loss of accuracy in the generation of keyboard events might be due to the higher load imposed by online applications running on a user agent, as compared to a simple Logkeys X-Window application. In any case, all the errors are below a millisecond, well below the resolution of standard DOM events (i.e., the `timestamp` property of DOMTimeStamp type, an `unsigned long long` aimed to store a value in milliseconds). The mean errors and standard deviations observed in the time stamps of DOM events are also below a millisecond in all cases. These results confirm the validity of this setting to collect user interaction data from online applications with high timing requirements.

Table 4 shows the results, in milliseconds, of the tests conducted with Mozilla Firefox. As can be seen, the results are very similar to those from the previous test, suggesting that the

Table 2 Timing errors in Study 1 (in milliseconds)

Interval	*Metronome LKM*		*Logkeys X-Window*	
	M	*SD*	*M*	*SD*
1	– 0.00000007	0.004660847	0.00000120	0.058875759
5	– 0.00000995	0.004499370	0.00410820	0.112822616
10	0.00000003	0.000192422	– 0.00083820	0.004214332
50	– 0.00000093	0.001032860	0.00054680	0.025032313
100	– 0.00000555	0.000930028	0.00459850	0.094976506
500	– 0.00001239	0.001294279	– 0.00349370	0.009127118
1,000	– 0.00000129	0.001002863	– 0.00017770	0.046649194

Table 3 Timing errors in Study 2 with Google Chrome 17 (in milliseconds)

Interval	*Metronome LKM*		*Google Chrome*	
	M	*SD*	*M*	*SD*
1	– 0.00139985	0.005827300	0.00000000	0.034642748
5	0.00000803	0.030747087	– 0.00070000	0.041227175
10	– 0.00000674	0.014677891	0.17050000	0.409690779
50	0.00000695	0.015569002	0.40250000	0.644309913
100	0.00000047	0.013899963	– 0.00010000	0.156532553
500	– 0.00000002	0.038588317	0.01180000	0.221958750
1,000	0.00000572	0.012574420	0.00000000	0.345560329

Table 4 Timing errors in Study 2 with Mozilla Firefox 10 (in milliseconds)

Interval	*Metronome LKM*		*Mozilla Firefox*	
	M	*SD*	*M*	*SD*
1	– 0.0030139	0.005213274	– 0.00050000	0.084264228
5	– 0.00000209	0.011312149	0.00010000	0.010000000
10	0.00000035	0.024555596	0.00050000	0.110909784
50	0.00000528	0.028443645	0.00000000	0.000000000
100	0.00000495	0.014243673	0.00000000	0.000000000
500	– 0.00000114	0.046871384	0.00000000	0.000000000
1,000	0.00000114	0.018916281	0.00000000	0.000000000

system loads imposed by Google Chrome and Mozilla Firefox do not differ substantially. Regarding the data collected with the Web application, there seems to be no measurement error for intervals longer than 10 ms, given the low resolution (milliseconds) of standard DOM events time stamps. Below 10 ms, we observed isolated timing errors with means below a microsecond and standard deviations below a millisecond. This confirms the results obtained with Google Chrome 17 and supports the validity of standard DOM events time stamps to collect user interaction data with standard Web applications.

Study 3

The results of Studies 1 and 2 allow us to conclude that, in principle (and setting aside the potential limitations of hardware; see Plant, 2016, and van Steenbergen & Bocanegra, 2016, for a discussion of this issue), it is possible to collect user interaction data with standard Web technologies with precision and accuracy below a millisecond. However, these results also reveal a limitation in the collection of reaction times with standard DOM events. Their low (millisecond) resolution can mask small variations at lower levels. It would be advisable to extend their resolution to microseconds or even nanoseconds. In fact, some developers of JavaScript projects related to time measurement (e.g., jsPerf, Benchmark.js) have decided to leverage the timing features of a Java applet provided with the nanosecond-resolution `nanoTime` function or, alternatively, to rely on proprietary extensions like `chrome.Interval`, which returns the number of microseconds since January 1, 1970.

Fortunately, the High Resolution Time API renders these alternative solutions unnecessary (Mann, 2012). This API implements a monotonically growing time function with accuracies below a millisecond (thanks to the `DOMHighResTimeStamp` type, which represents values in milliseconds with a precision of microseconds) and is immune to problems arising from clock drifts in multiprocessor systems or changes in the system clock.

Given the advantages of this method, we thought that it would be advisable that DOM events registered their time stamps with the `DOMHighResTimeStamp` type, so that these values could be compared to those obtained with newer APIs that rely on this type of time stamps, such as the High Resolution API (Mann, 2012) or the API for the time control of procedural animations (Robinson & McCormack, 2015), among others. Because of this, we submitted a request to implement this improvement in version 4 of DOM Events to the W3C's DOM Events Working Group (see http://lists.w3.org/Archives/Public/www-dom/2012AprJun/0077.html and http://lists.w3.org/Archives/Public/www-dom/2012AprJun/0092.html).

Our request is still pending approval but it has already been implemented in latest versions of Google Chrome and Mozilla Firefox. The goal of Study 3 was to benchmark a JavaScript Web application running on more recent browsers with high-resolution time stamps for events (Google Chrome 58 and Mozilla Firefox 54) using Metronome LKM.

Method

Apparatus, materials, and procedure As in Studies 1 and 2, we installed Metronome LKM on an Ubuntu Linux vanilla distribution unconnected to the Internet and isolated from external sources of asynchronous events. On this setting, we run a JavaScript Web application that relies on the high-resolution time stamps of DOM Events to log the time elapsed from the beginning of the execution of the Web application (`navigationStart`) to the moment when the keyboard event is registered (i.e., the key-press event). The procedure was otherwise identical to Studies 1 and 2.

Results and discussion

Table 5 summarizes the results obtained with Google Chrome 58. All the values refer to measurement errors in milliseconds. As in the previous studies, the mean errors obtained by Metronome LKM were close to a nanosecond, and their standard deviations were below a millisecond. Despite the numerous upgrades in Google Chrome between Studies 2 and 3 (i.e., from version 17 to 58), Metronome LKM behaved similarly. The mean errors and standard deviations in the time stamps of high-resolution DOM events were again below the millisecond in all cases, confirming the validity of this setting to collect user interaction data from online applications with high timing requirements.

To assess how system load can affect the accuracy and precision of Metronome LKM, we repeated this study enabling the CPU throttling feature. Google Chrome's CPU throttling allows Web developers to simulate the execution

Table 5 Timing errors in Study 3 with Google Chrome 58 (in milliseconds)

Interval	*Metronome LKM*		*Google Chrome*	
	M	*SD*	*M*	*SD*
1	– 0.0000002315	0.010297074697692	0.000000000000000	0.000000000000835
5	– 0.0000030569	0.029049538686114	0.000000000000000	0.074836889673427
10	– 0.0000008563	0.030050350461519	0.000000000000002	0.273143663199602
50	– 0.0000009461	0.020421222608876	0.000000000000008	0.014142842783138
100	0.0000066783	0.019588973776652	0.000000000000014	0.000000000052873
500	– 0.0000002279	0.016793519432944	0.000000000000101	0.000000000011067
1,000	– 0.0000003735	0.010597582993198	0.000000000000216	0.000000000013466

of a Web application in low-resource or overloaded computers. This feature can be set to a range of values from 2x to 20x, representing increasing levels of overload. We repeated the procedure of the present study while setting this variable to 20x. Table 6 shows the results of the tests conducted with 20x CPU throttling on Google Chrome 58. Interestingly, these results do not differ substantially from those gathered without CPU throttling. (If anything, they are slightly better, because in the 10-ms interval condition on Google Chrome 58 without throttling, 7 out of 10,000 key events were processed 1–3 ms late, whereas this did not happen in the tests with throttling.) This led us to think that Metronome LKM can be used to generate key events accurately and precisely even under considerable CPU overload.

Table 7 shows the results of the tests conducted with Mozilla Firefox 54. As can be seen, the results from Metronome LKM are very similar to those in the previous tests. Regarding the data collected with the Web application, this Mozilla Firefox version behaves significantly worse than the others. It cannot properly register all key events in the most demanding case (1-ms interval). In terms of precision, we found submillisecond precision in the 1-, 5-, 10-, 50-, and 1,000-ms interval tests. However, the precisions of tests with 100- and 500-ms intervals were 1.666 and 1.988 ms, respectively. These suboptimal results might be explained because version 54 is the first Mozilla Firefox version (and the last available stable version at this moment) to provide high-resolution event time stamps, and this feature might not have been thoroughly tested yet.

Study 4

In Studies 1–3, we used Metronome LKM to explore the precision of the new DOM events in the latest versions of Google Chrome and Mozilla Firefox. Assessing their accuracy would require an analysis of the absolute values of the time stamps, instead of the relative difference between consecutive keyboard events. This is a daunting challenge in new systems, due to the variety of timing sources used by different timing functions. Both Metronome LKM and `dmesg` rely on a monotonic timing source not synchronized with the system clock. This timing source monotonically updates the value of a time counter relative to the system uptime. Therefore, the system clock can potentially drift away with respect to this monotonic timing function. In fact, it is easy to check this by running the following commands in Linux with root permission: `echo test > /dev/kmsg && dmesg -T | tail -1 && date`. If there is no drift between the two timing sources, both dates should be identical. However, if we suspend the

Table 6 Timing errors in Study 3 with Google Chrome 58 at 20x CPU throttling (in milliseconds)

Interval	*Metronome LKM*		*Google Chrome 20x*	
	M	*SD*	*M*	*SD*
1	– 0.0000000805	0.001502703393211	0.000000000000000	0.000000000005242
5	– 0.0000001081	0.016242222684300	0.000000000000001	0.000000000004961
10	– 0.0000011512	0.014242128505942	0.000000000000000	0.000000000007990
50	0.0000029802	0.015567334193438	0.000000000000006	0.000000000029523
100	0.0000030432	0.016285437520376	0.000000000000005	0.004472359574564
500	– 0.0000000095	0.015682590518455	0.000000000000074	0.000000000005470
1,000	0.0000000064	0.014487621290040	– 0.000000000000030	0.000000000011267

Table 7 Timing errors in Study 3 with Mozilla Firefox 54 (in milliseconds)

Interval	*Metronome LKM*		*Mozilla Firefox*	
	M	*SD*	*M*	*SD*
1	0.0000022604	0.019428239233383	1.933267773400000	0.536804231230342
5	0.0000065850	0.036059731861193	– 0.000576063500000	0.642599968616566
10	0.0000025690	0.043567016249743	– 0.000659114599999	0.832928759893024
50	0.0000058274	0.012179655831148	– 0.000212510299996	0.591939754346651
100	0.0000004561	0.011726603144952	– 0.000323846800007	1.665740962872240
500	0.0000000343	0.012328960555666	– 0.000397035300001	1.987785052628540
1,000	0.0000006020	0.013258050157191	– 0.000313978800020	0.490275936869454

computer for several minutes and repeat the test, the drift would be evident. In any system that updates the system clock automatically through NTP or similar protocols, there is a good chance that the system clock will drift away from the monotonic timing source of the kernel.

Regarding DOM events time stamps, the standard events use the system clock as the timing source and, consequently, their values can be directly compared to the system clock and are subject to the same drift. On the contrary, DOM events with high-resolution time stamps return a value relative to the beginning of the execution of the Web application (`navigationStart`) and can only be compared to the system clock in the absence of drift (i.e., `navigationStart` + `event.timeStamp` should be equivalent to the value returned by `Date`).

This being said, would it be possible to compare the time stamps gathered by Metronome LKM with the DOM events' high-resolution time stamps? As can be seen in Fig. 1, the time stamps of Metronome LKM are offset relative to the system boot time (btime). The corresponding time in the system clock can be obtained with the command `grep btime /proc/stat`. Regrettably, this value has a precision of seconds, which limits the accuracy of any comparison between btime and the system clock. The same problem applies to the starting point of the browser time stamps. The `navigationStart` property stores the value of the wall clock in milliseconds at the beginning of execution of the Web application. However, this value cannot be directly compared to btime, due to differences in resolution and to the potential drift of the system clock. Additionally, the fact that `navigationStart` returns values in milliseconds, whereas high-resolution time stamps work in microseconds, implies that any comparison between these time stamps and other events storing time stamps from the system clock can be made only at the millisecond level. In spite of these hindrances, we decided to conduct a final study to analyze the accuracy of the keyboard events generated by Metronome LKM and registered by Google Chrome 58.

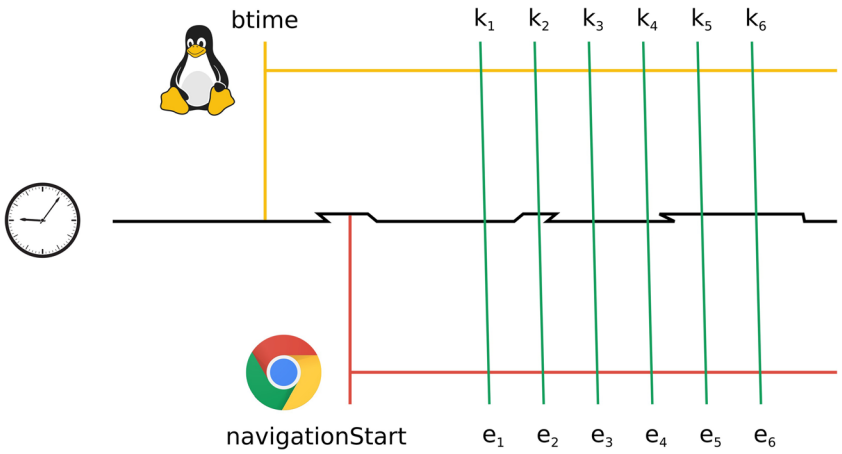

Fig. 1 The three time lines involved in the set of studies presented in this article. The middle time line represents the system clock (often known as the "wall clock"), subject to dynamic adjustments and time drift. Top time line represents the monotonic clock of the Linux kernel: It starts at btime, and all subsequent time stamps are offsets relative to this starting point. The bottom line represents the monotonic clock of the High Resolution Timing API of the browser: It starts at navigationStart, and all subsequent time stamps are offsets relative to this starting point. Key events generated by Metronome are marked as k_n, and the corresponding input events (e_n) are registered slightly later by the browser.

Method

Apparatus, materials, and procedure We modified the source code of Metronome LKM to provide not only the time stamp gathered from Linux kernel's monotonic time source, but also from the system clock (via the `getnstimeofday` timing function). Then, we installed this modified version of Metronome LKM on an Ubuntu Linux vanilla distribution unconnected to the Internet and isolated from external sources of asynchronous events. On this setting, we run a JavaScript Web application that relies on the high-resolution time stamps of DOM Events to log the time elapsed from the beginning of the execution of the Web application (`navigationStart`) to the moment when the keyboard event is registered (i.e., the keypress event). The procedure was similar to Studies 1–3, but we reduced the number of samples collected from 10,000 to 50 because the precision of the software had already been assessed in the previous studies, and this study was aimed at determining the delay between the generation of the key event by Metronome LKM and the time stamp gathered by the browser.

Results and discussion

For every keyboard event generated, we calculated its UNIX time stamp (the time elapsed from the 1st of January, 1970), both in the kernel log generated by Metronome LKM and in the data file generated by the Web application running on Google Chrome 58. Then we calculated the difference between the browser and kernel time stamps and found a 0.779 ± 0.343-ms difference for all intervals: (a) 1,000 ms (M = 0.770191, SD = 0.031244); (b) 500 ms (M = 1.091819, SD = 0.010752); (c) 100 ms (M = 0.838900, SD = 0.022353); (d) 50 ms (M = 1.109455, SD = 0.024969); (e) 10 ms (M = 1.087820, SD = 0.023810); (f) 5 ms (M = 0.202062, SD = 0.031899); and (g) 1 ms (M = 0.354179, SD = 0.010953).

To assess the accuracy of reaction time measures, one needs to collect the time stamp of the moment when the original event was created, not the time stamp of the moment when it was logged. As can be seen in the raw data from our studies, available at https://osf.io/6dv3e, there is a noticeable delay between the time stamp recorded by Metronome LKM and the one recorded at the kernel's ring buffer (e.g., in `[ 6287.072289] metronome: Key event (57, 6287064130600ns`, there is a 8.1584-ms difference between the two time stamps). This caveat applies to the processing of DOM event time stamps in Web applications or other types of software (i.e., avoiding calling the `performance.now` function inside the event handler, and instead using the event's time stamp defined at the time of creating the event itself).

Conclusions and outlook

In the present article we have introduced Metronome LKM, a new tool to benchmark the accuracy and precision of reaction time measures taken in experimental software running under GNU/Linux. This tool allows researchers to generate key-press events using high-resolution timers and compare the time stamps collected by the experiment software with those gathered by Metronome LKM (with nanosecond resolution). After testing the performance of LKM (Study 1), we have used it to benchmark the precision (Studies 2 and 3) and accuracy (Study 4) of a JavaScript application running under several versions of Google Chrome and Mozilla Firefox. Our results confirm that the DOM event time stamps can be used to collect user interaction data with standard Web applications. Furthermore, they also highlight the value of Metronome LKM as a benchmarking tool. All the data sets used in the present studies are available under a free license at https://osf.io/6dv3e.

To confirm whether the present results can be generalized to other versions of Google Chrome and Mozilla Firefox and operating systems other than Linux, we have run the main JavaScript benchmarking suites in Google Chrome and Mozilla Firefox running under Ubuntu Linux 16.04.2 and Microsoft Windows 10 Pro. Both for Google (Octane 2.0: 25068 in Linux vs. 25102 in Windows; JetStream: 140.30 ± 7.0520 in Linux vs. 133.72 ± 2.7159 in Windows; Kraken: 1251.1 ± 0.7% in Linux vs. 1310.8 ± 1.5% in Windows) and Mozilla Firefox (Octane 2.0: 25266 in Linux vs. 25167 in Windows; JetStream: 143.60 ± 9.8593 in Linux vs. 133.35 ± 7.0495 in Windows; Kraken: 1257.7 ± 4.7% in Linux vs. 1351.5 ± 2.4% in Windows), we obtained very similar results in Linux and Windows. These converging results are consistent with the development model adopted for these two browsers, as both of them rely on a common base of source code for their implementation in each operating system.

It is worth noting that Metronome LKM can also be used in combination with other benchmarking methods. For instance, as explained in the Introduction, some studies have assessed the reliability of reaction time measures using devices that can detect changes in the screen and respond to them sending a signal back to the experimental computer (Häusler et al., 2007; Neath et al., 2011; Rorden & Hanayik, 2014; Schubert et al., 2013). This approach is an excellent means to evaluate whole experimental settings, but its results cannot inform us about the specific sources of noise that might affect the data. Any problem in the accuracy or precision of reaction time measures could be due either to the experimental software, hardware, or an interaction between both. Metronome LKM can be used in combination with these methods to isolate the specific contribution of experimental software to the overall amount of noise in reaction times, providing a useful baseline for the assessment of any additional noise induced by hardware.

To sum up, Metronome LKM provides a simple means to simulate user keystrokes precisely and accurately. Its open source license enables researchers and developers to audit, correct, and adapt it as they see fit. However, it is important to remember that software benchmarking is aimed to find unforeseen errors and improve software, not to validate experimental setups. Researchers must test their whole setups (including hardware and software) before conducting experiments, using more comprehensive methods, such as pilot experiments with participants or using specialized hardware.

Author note Support for this research was provided by Departamento de Educación, Universidades e Investigación of the Basque Government (Grant No. IT1078-16). M.A.V. was supported by Programa de Atracción de Talento Investigador de la Comunidad de Madrid (Grant 2016-T1/SOC-1395). The authors declare no conflict of interest in the publication of this study.

References

Badle, S., Bakken, J., Barantsev, A., Beans, E., Berrada, D., Bevan, J., . . . Wagner-Hall, D. (2012). Selenium—Web Browser Automation. Retrieved from http://seleniumhq.org.

Bhalla, M., & Bhalla, A. (2010). Comparative study of various touchscreen technologies. *International Journal of Computer Applications*, *6*, 12–18.

Bovet, D.P., & Cesati, M. (2005). *Understanding the Linux kernel* (3rd ed.). O'Reilly.

Brainard, D. H. (1997). The Psychophysics Toolbox. *Spatial Vision*, *10*, 433–436. doi:https://doi.org/10.1163/156856897X00357

Chapweske, A. (2003). *The PS/2 mouse/keyboard protocol.* Retrieved from www.computer-engineering.org/ps2protocol.

Crosbie, J. (1990). The Microsoft mouse as a multipurpose response device for the IBM PC/XT/AT. *Behavior Research Methods*, *22*, 305–316.

Damian, M. F. (2010). Does variability in human performance outweigh imprecision in response devices such as computer keyboards? *Behavior Research Methods*, *42*, 205–211. doi:https://doi.org/10.3758/BRM.42.1.205

de Leeuw, J. R. (2015). jsPsych: A JavaScript library for creating behavioral experiments in a web browser. *Behavior Research Methods*, *47*, 1–12. doi:https://doi.org/10.3758/s13428-014-0458-y

Eichstaedt, J. (2001). An inaccurate-timing filter for reaction time measurement by JAVA applets implementing Internet-based experiments. *Behavior Research Methods, Instruments, & Computers*, *33*, 179–186.

Häusler, J., Sommer, M., & Chroust, S. (2007). Optimizing technical precision of measurement in computerized psychological assessment on Windows platforms. *Psychology Science*, *49*, 116–131.

Henninger, F., Mertens, U. K., Shevchenko, Y., & Hilbig, B. E. (2017). lab.js: Browser-based behavioral research. doi:https://doi.org/10.5281/zenodo.597045

Keller, F., Gunasekharan, S., Mayo, N., & Corley, M. (2009). Timing accuracy of web experiments: A case study using the WebExp software package. *Behavior Research Methods*, *41*, 1–12. doi:https://doi.org/10.3758/BRM.41.1.12

Mathôt, S., Schreij, D., & Theeuwes, J. (2012). OpenSesame: An open-source, graphical experiment builder for the social sciences. *Behavior Research Methods*, *44*, 314–324. doi:https://doi.org/10.3758/s13428-011-0168-7

Mann, J. (2012). High Resolution Time (W3C candidate recommendation 22 May 2012). Retrieved from www.w3.org/TR/2012/CR-hr-time-20120522/.

Neath, I., Earle, A., Hallett, D., & Surprenant, A. M. (2011). Response time accuracy in Apple Macintosh computers. *Behavior Research Methods*, *43*, 353–362. doi:https://doi.org/10.3758/s13428-011-0069-9

Pearce, J. M. (2012). Building research equipment with free, open-source hardware. *Science*, *337*, 1303–1304.

Peirce, J. W. (2007). PsychoPy—Psychophysics software in Python. *Journal of Neuroscience Methods*, *162*, 8–13. doi:https://doi.org/10.1016/j.jneumeth.2006.11.017

Peirce, J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics. 2*, 10. doi:https://doi.org/10.3389/neuro.11.010.2008

Pelli, D. G. (1997). The VideoToolbox software for visual psychophysics: Transforming numbers into movies. *Spatial Vision*, *10*, 437–442. doi:https://doi.org/10.1163/156856897X00366

Pixley, T. (2000). Document object model events. Retrieved from www.w3.org/TR/DOM-Level-2-Events/events.html

Plant, R. R. (2016). A reminder on millisecond timing accuracy and potential replication failure in computer-based psychology experiments: An open letter. *Behavior Research Methods*, *48*, 408–411. doi:https://doi.org/10.3758/s13428-015-0577-0

Plant, R. R., Hammond, N., & Turner, G. (2004). Self-validating presentation and response timing in cognitive paradigms: How and why? *Behavior Research Methods, Instruments, & Computers*, *36*, 291–303. doi:https://doi.org/10.3758/BF03195575

Plant, R. R., Hammond, N., & Whitehouse, T. (2003). How choice of mouse may affect response timing in psychological studies. *Behavior Research Methods, Instruments, & Computers*, *35*, 276–284. doi:https://doi.org/10.3758/BF03202553

Plant, R. R., & Turner, G. (2009). Millisecond precision psychological research in a world of commodity computers: New hardware, new problems? *Behavior Research Methods*, *41*, 598–614. doi:https://doi.org/10.3758/BRM.41.3.598

Reimers, S., & Stewart, N. (2007). Adobe Flash as a medium for online experimentation: A test of reaction time measurement capabilities. *Behavior Research Methods*, *39*, 365–370. doi:https://doi.org/10.3758/BF03193004

Robinson, J., & McCormack, C. (2015). Timing control for script-based animations (W3C Working Group Note, 22 September, 2015). Retrieved from www.w3.org/TR/2015/NOTE-animation-timing-20150922/

Rorden, C., & Hanayik, T. (2014). StimSync: Open-source hardware for behavioral and MRI experiments. *Journal of Neuroscience Methods*, *227*, 90–99.

Schneider, W., Eschman, A., & Zuccolotto, A. (2002). E-Prime user's guide. Pittsburgh, PA: Psychology Software Tools.

Schubert, T. W., D'Ausilio, A., & Canto, R. (2013). Using Arduino microcontroller boards to measure response latencies. *Behavior Research Methods*, *45*, 1332–1346. doi:https://doi.org/10.3758/s13428-013-0336-z

Segalowitz, S. J., & Graves, R. E. (1990). Suitability of the IBM XT, AT, and PS/2 keyboard, mouse, and game port as response devices in reaction time paradigms. *Behavior Research Methods, Instruments, & Computers*, *22*, 283–289.

van Steenbergen, H., & Bocanegra, B. R. (2016). Promises and pitfalls of Web-based experimentation in the advance of replicable psychological science: A reply to Plant (2015). *Behavior Research Methods*, *48*, 1713–1717. doi:https://doi.org/10.3758/s13428-015-0677-x