HLISA: towards a more reliable measurement tool*

Daniel Goßen
Radboud University Nijmegen
Netherlands

Hugo Jonker
Open University Netherlands,
Radboud University Nijmegen
firstname.lastname@ou.nl

Stefan Karsch
TH Köln, Germany
firstname.lastname@th-koeln.de

Benjamin Krumnow†
Open University Netherlands,
TH Köln, Germany
firstname.lastname@th-koeln.de

David Roefs
Radboud University Nijmegen
Netherlands

ABSTRACT

Automated browsers (web bots) are an invaluable tool for studying the web. However, research has shown that web bots can be distinguished from regular browsers and that they may be served different content as a consequence. This undermines their utility as a measurement tool. So far, three methods have been used to detect web bots: browser fingerprint, order of site traversal, and aspects of page interaction.

While site traversal depends on the study being executed, the other two aspects can be controlled in a generic fashion. Whereas identifiability of web bot fingerprints has been studied in the past, *how* to alter the fingerprint has received less attention. In this paper, we study which method to alter the fingerprint incurs the least side effects. Secondly, we provide an initial investigation of how the interaction API of Selenium differs from human interaction. We incorporate the latter results into HLISA, an API that simulates interaction like humans. Finally, we discuss the conceptual arms race between simulators and detectors and find that conceptually, detecting HLISA requires modelling human interaction.

CCS CONCEPTS

• **Security and privacy** → *Browser security*; • **Information systems** → *Browsers*; *Data extraction and integration.*

KEYWORDS

web bot detection, web studies, reliability, behavioural recognition, browser fingerprinting

ACM Reference Format:
Daniel Goßen, Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and David Roefs. 2021. HLISA: towards a more reliable measurement tool. In *ACM Internet Measurement Conference (IMC '21), November 2–4, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3487552.3487843

*Authors in alphabetical order
†Corresponding author

IMC '21, November 2–4, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9129-0/21/11…$15.00
https://doi.org/10.1145/3487552.3487843

1 INTRODUCTION

Automation is an inevitable requirement for large-scale web studies. Over the years, tools used by such studies have matured, from simple command-line HTTP clients (e.g., wget), via extensive HTTP libraries (e.g., BeautifulSoup), to headless browsers (e.g., PhantomJS). However, compared to conventional browsers, such tools have significant drawbacks, lacking features and rendering capabilities. A more recent trend addresses this by using automation frameworks to automate regular browsers. Interestingly, Jonker et al. [12] found that the browser fingerprint of such browsers deviates from their non-automated counterparts. They also found that websites use these differences to serve different content to web bots than to regular browsers. Even though their study found that only a few sites employed such measures, these tactics undermine the fundamental assumption of web studies: that the measuring tool used encounters the same web as regular browsers would.

Serving different content hinges on recognising web bots as such. Three avenues for web bot detection have been identified: browser fingerprinting (e.g., [12, 36]), site traversal (e.g., [33]), and interaction characteristics (e.g., [3]). Web studies that want to minimise the error due to web bot detection should account for all three aspects. Crucially, mitigating site traversal – the path an automated browser takes over a website – cannot be solved generically, as such paths depend on the study being executed. However, neither browser fingerprint nor interaction characteristics are (typically) study-dependent. Both aspects can thus be generically addressed.

In this paper, we investigate these two aspects. More particularly, previous studies have found what properties to change about the browser fingerprint: eliminating the `webdriver` property suffices for most contemporary detection approaches, while methods to determine the full fingerprint needed are available [12, 27]. However, *how* to alter the browser fingerprint has not been as comprehensibly studied. We investigate various ways to spoof aspects of a browser fingerprint using JavaScript and elucidate the side effects inherent in each approach. Secondly, previous studies and online advice usually focused on changing one specific form of interaction to reduce the distinctiveness of its automation. In contrast, we offer a comprehensive study on identifying and reducing distinctiveness for all types of user interaction observable via JavaScript.

Contributions. The contributions of this paper are:

(1) We compare known means to spoof identifiable properties in web bot frameworks. We are the first to investigate side effects that allow the detection of a spoofing attempt. Moreover, we test the effectiveness of best method in the wild.

(2) We experimentally establish the recognisability of the Selenium interaction API and provide a proof-of-concept interaction library, HLISA, to address the identified shortcomings.
(3) We model the action-reaction cycle between detectors and simulators as an arms race. This clarifies their limitations, and how to escalate their respective capabilities. Moreover, the model enables a legal analysis of detection approaches vs privacy regulations.

2 RELATED WORK

Fingerprinting for web bot detection. It is well established that page traversal and page interaction can be used to detect web bots [6, 22, 33]. A relatively new approach is to identify browser properties that are unique to web bots in contrast to conventional browsers used by real humans. Such unique properties enable distinguishing a web bot from human visitors on the very first page visited, without any interaction. Early insights about identifiable properties were reported in blogs by security researchers [31, 35]. Jonker et al. [12] introduced the first systematic approach to find identifiable properties. They used browser fingerprinting to identify differences between web bots and the conventional browsers these bots were based on. Schwarz et al. [27] introduced JavaScript template attacks. They traversed the JavaScript object tree hierarchy to create a template of a client's setup. Where traditional browser fingerprinting makes use of a predefined set of known characteristics, the template attack method traverses the DOM to find differences.

Jonker et al. [12] investigated whether spoofing identifiable properties in a Selenium-controlled Chrome browser affects blocking on websites and found that this indeed occurs. Vastel et al. [36] systematically tested the resilience of bot detectors to spoofing and found that detectors highly depend on the `webdriver` attribute. Jueckstock et al. [13] experimented with a puppeteer spoofing extension to hide properties in Chrome. They compared measurements between their hidden Chrome and a headless Chrome browser. They found significant differences in third-party traffic and JavaScript execution. Spoofing techniques have also often be used as a defence against browser fingerprinting. Such spoofing can be performed either directly in JavaScript [7, 34] or on the browser level [14, 19].

Detecting artificial interaction. Many studies have used interaction as a means to re-identify individuals. A common use case is to use interaction as a complement to passwords. The focus of these studies, therefore, mostly concerns typing rhythms as a secondary authenticator [24]. This idea was also adapted to mobile phones. Giuffrida et al. [10] used touch events and enhanced this with data from smartphone sensors (e.g., accelerometer and orientation sensor). In contrast, studies have also examined automating interaction to perform credential stuffing or brute force attacks. Such works challenge the security of interaction detection systems by imitating human interaction with bots. For example, the study by Serwadda and Phoha [29] demonstrates such an attack by using a large-scale data set with keystrokes from real users to simulate human typing. Similarly, Serwadda et al. [30] brought this attack to mobile phones by leveraging a robotic arm to mimic gestures.

Bots are used in other circumstances to circumvent expectations. This is particularly egregious in online games (e.g., aim bots) and online discussion fora (e.g., spam bots). The academic community has investigated these aspects at great length. For example, mouse button press and release events were used by Barik et al. [2] to detect automated clients in web browser games. Limited variety of player interaction was used by Gianvecchio et al. [9] to identify bots in the context of an MMO. Chu et al. [3] used keystroke dynamics and mouse movement to identify spam bots on blogs. Detection of such bots is also a hot topic outside academia. As such bots cause significant annoyance for users and maintainers/moderators, a plethora of tools, blog posts and other non-peer reviewed discussions exist that attempts to identify such bots. On the other hand, such bots can offer significant benefits to their users, who actively discuss ways to introduce more realistic interaction into them (cf. comparison in Appendix G).

3 EVADING FINGERPRINT-BASED DETECTION

Browser fingerprinting allows websites to detect bots without any interaction on a page. Hence, to prevent bot detection, the first defence is to hide a web bot's identifiable properties.

There are two approaches to changing browser properties: modifying the browser's source code and overriding properties at run time (e.g., via JavaScript). Both have their merits and disadvantages. Overriding based on the built-in functionality of the JavaScript API can be applied dynamically, even for properties created at run time. Furthermore, it is easy to deploy, as JavaScript code is directly injected into a web page and is cross-platform compatible. One downside is that side effects may occur due to JavaScript quirks. Moreover, such modifications can break websites, as already found for some privacy extensions [5].

In contrast, browser level patches of properties avoid the introduction of such side effects. However, adjusting the browser source code adds considerable overhead. First, adjustments must be maintained for newly appearing browser versions. Second, the browser build process may be a challenge for some users. Lastly, a browser level implementation binds a solution to a single platform. However, it is currently unclear if JavaScript level patching can be made without thwarting web measurements.

In the remainder of this section, we explore approaches in JavaScript for property spoofing. We use our insights to build an extension for OpenWPM, that can hide detectable properties. Finally, we test our implementation on 1,000 websites.

3.1 JavaScript-based spoofing methods

To select spoofing methods suitable for our evaluation, we consider approaches used earlier in the literature and approaches adopted by popular browser extensions. In more detail, we conducted source code reviews of spoofing extensions available for Firefox [17, 18, 25, 26, 28], academic work to fight browser fingerprinting [4, 7, 19, 34] and research to block ads [32]. This led us to the following methods:

(1) **defineProperty** is a built-in function of JavaScript objects to directly set or alter an object's property.
(2) **__defineGetter__** overrides a `getter`-function allowing us to return a specific value without changing it. Note that this function was deprecated by Mozilla.

Table 1: Detectable side effects by spoofing methods

Side effect	Spoofing method 1	2	3	4
Incorrect order of `navigator` properties	×	×		
Modified `navigator._length`	×	×		
New `Object.keys(navigator)`	×	×		
Defined `navigator.__proto__.webdriver`			×	
Unnamed `window.navigator` functions				×

(3) **setPrototypeOf** sets a new prototype for an object which provides control over the object's properties.
(4) **Proxy objects** allow to re-define the behaviour of an object by wrapping it with a proxy object.

We tested each method to spoof `navigator.webdriver` property to return *false* within Firefox. Note that this is an easy-to-use property available by convention [37] and plays a crucial role in the identification of WebDriver-controlled user agents [36]. To check for the occurrence of side effects of each method, we use JavaScript template attacks by Schwarz et al. [27] and the modified fingerprint library by Jonker et al. [12].

Table 1 list the four evaluated methods along with their side effects. Interestingly, none of the previously applied methods was side-effect free in our measurement. For methods 1 and 2, we observe that each attempt to spoof a property increments the `navigator.length` property. Spoofing the length property in this manner is insufficient, as its original value remains in the prototype chain. In addition, we find a change in the order of items when iterating through properties of the navigator properties. This reveals which property has been overwritten. In a regular Firefox browser, the `webdriver` attribute is enumerable but disappears from the listing when calling `Object.keys(navigator)` in our overwritten version. However, it is possible to remedy this by setting the enumerable property to *true*. The third method does not include the previous method's side effects but is inherently detectable. In regular Firefox, the chain `__proto__` is not defined for the `webdriver` attribute but needs to be set in order to execute this method. We find that wrapping the navigator object is detectable by calling the `toString` function for the last approach. As shown in Listing 1, the result of this overriding leads to missing function names.

```
//Call of a toString function of a built-in method
window.navigator.toString.toString();

// Output in a regular Firefox browser
  "function toString() {
     [native code]
   }"

// Output after shadowing methods via proxy objects
  "function () {
    [native code]
   }"
```

Listing 1: Detectability of JavaScript spoofing with on proxies objects.

We conclude that JavaScript proxy objects appear to be most suitable for spoofing. While an adversarial website could spot a wrapped navigator object, it does not know what property was changed when applying this approach to multiple properties. Further, benign web users may apply the same techniques through extensions, e.g., userAgent spoofers, privacy extensions, and such. Still, the effectiveness of this approach must be evaluated on real world websites.

Table 2: Results from the screenshot evaluation.

Response	sites (1)	sites (2)	visits (1)	visits (2)
total	**921**	**921**	**7,230**	**7,221**
missing ads	7	3	56	10
– no ads	5	1	40	4
– less ads	2	2	16	6
blocking/CAPTCHAs	8	1	49	3
frozen video element(s)	1	0	8	0

Results for crawler OpenWPM (1) and OpenWPM+extension (2).

3.2 Evaluation

We developed a browser extension to spoof the *webdriver* property in OpenWPM clients based on our selected method. To evaluate our extension in the field, we run OpenWPM with/without our extension on two different machines. We use a consistent setup with OpenWPM v.0.13.0 and run the Firefox browsers in headful mode. We then let both machines visit the same set of websites simultaneously. Our website set consists of a random selection of 1,000 sites taken from the Top 10K websites of the Tranco list [15]. As our experiment could be influenced through web dynamics (bidding processes, content updates, etc.) as well as blocking through suspicious IP ranges (e.g., university or cloud-based IP addresses [11, 38]), we take precautions. First, we use distinct residential IP addresses and second, we run 8 browsers instances simultaneous per machine. This provides us with a baseline to average out variations.

Incidence of blocking. We check if our extension mitigates bot detection while not breaking websites. To measure incidences of detection, we review screenshots and count the occurrence of blocking pages, CAPTCHAs, visible error messages and HTTP response status codes that occur only for one machine. In addition, we evaluate if there is missing content (such as ads). We choose to use visual responses as these allow definitive attribution to bot detection, while other measurable aspects (e.g., cookies or HTTP traffic) do not. Note that not all web sites with bot detectors react visually to automated visitors [13]. As such, this check is only an approximation to determine the effect of using the spoofing implementation.

Table 2 breaks down our results of screenshot evaluation separated into reached sites and successful visits. We see that the number of sites with visible signs of bot detection is low. All observed differences combined occur on only 16 (1.7%) sites for the regular OpenWPM client. We find that spoofing significantly reduces this effect. In fact, we see only one site that deploys blocking against our extended OpenWPM version for a smaller subset of visits.

To identify blocking at HTTP level, we look at status codes in HTTP responses. We separated these by first and third-party responses. We further use Wilcoxon Matched-Pairs signed-Rank

Test with a confidence interval of 95% to test for significance. Our results show only a notable difference in first-party errors, with a significant decrease (p-value = 0.004) when using our extension. We find that this decrease is mainly due to responses with 403 (forbidden) and 503 (service unavailable) as shown in Appendix B. Both status codes can be related to the effect of bot detection.

Incidence of website breakage. Lastly, we look for deformed layouts, frozen elements, missing content and HTTP errors to spot if any website breakage occurred. Interestingly, we identified one site with a deformed layout and one site with an ever-loading video element, which hints at compatibility issues on these sites. Unfortunately, while the effect persisted when running the experiment on another machine, we failed to identify the root cause for these breaks.

4 RECOGNISING GENERATED INTERACTION

In this section, we examine how page interaction generated by OpenWPM is identifiable from non-automated clients. OpenWPM does not offer its own interaction API, but simply exposes the Selenium interaction API, which communicates via the WebDriver protocol with Firefox's browser engine (Gecko). A web page can monitor such interactions through JavaScript events.

To take the website perspective, we set up an experiment to measure the interaction of Selenium, ourselves, and naive approaches to improve Selenium's shortcomings. Based on the experiment results, we design and implement the *Human-Like Interaction Selenium API* (HLISA) to reduce the detectability of any web bot using the Selenium framework (such as OpenWPM).

4.1 Improving Selenium's interaction API

We performed several experiments to compare the interaction of Selenium with that of a regular human (ourselves). For that, we distinguish between fine-grained and course-grained actions. A fine-grained action is a single operation, such as a mouse button press. In contrast, a coarse-grained action consists of a chain of fine-grained actions. For example, clicking an element on a page could consist of three steps: moving the cursor, pressing the mouse button, and releasing it. We based our investigation on coarse-grained actions that are essential for typical web browsing activities (typing, clicking, mouse movement and scrolling). We found that Selenium's speed and precision far outstrip human capabilities. We created a new *Human-Like Interaction Selenium API* (HLISA) which addresses these concerns and provides more human-like interaction. Below, we discuss our findings and solutions to mouse movement, scrolling, clicking and typing in more detail. We refer to Appendix E for a detailed description of our setup to record interactions.

Mouse movement (Fig. 1). Mouse movement of a human has an initial acceleration, deceleration near the end of the trajectory, and moves in a jitterish curved trajectory. Selenium's interaction API, in contrast, moves at a uniform speed over a straight line. A naive solution would be to use a straightforward Bézier curve, but as is apparent from Fig. 1, this is still very artificial. In contrast, HLISA modifies a Bézier curve by starting with acceleration and ends with deceleration, over a jittery curve. We use the speed, acceleration and jitter of the mouse movement observed in the experiment as a baseline.

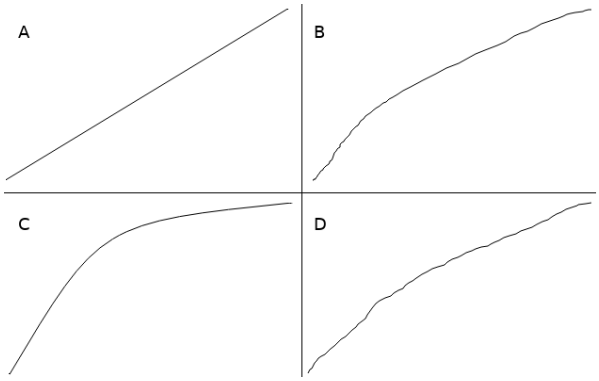

Figure 1: Cursor trajectories for: (A) Selenium, (B) human, (C) naive solution, (D) HLISA.

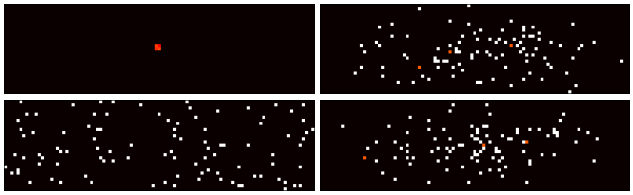

Figure 2: Experiment: distribution of mouse clicks of (top left) Selenium, (top right) humans, (bottom left) naive solution, (bottom right) HLISA.

Mouse clicks (Fig. 2). Selenium clicks perfectly in the centre of the HTML element. In contrast, clicks by humans are much more distributed but hardly ever in the centre. A naive approach would be to randomise the click location, e.g., using a uniform distribution (bottom left in Fig. 2). While this does improve over Selenium's default behaviour, it generates clicks in places humans never reach. In contrast, HLISA uses a normal distribution with parameters drawn from our experiment.

Scrolling. Our experiments with scrolling found that Selenium does not offer an API for scrolling. Its default method for scrolling lacks mouse wheel events and can scroll arbitrary long distances in one scroll event, as opposed to human scrolling via the mouse wheel. However, there is a plethora of other human interactions that trigger scrolling, including scroll bar, anchor links, arrow keys, space bar, search functionality, and Firefox's auto scroll. As such, detecting bots based on scrolling is challenging and there is no obvious, naive solution that does address Selenium's shortcomings. Nevertheless, HLISA extends the Selenium API with a function to simulate scrolling, which uses the default mouse wheel scroll distance (57 pixels), uses a normal distribution to incorporate short breaks, and incorporates a slightly longer break to account for moving one's finger to continue scrolling the mouse wheel.

Key presses. Key presses of Selenium have a negligible dwell time (time a key is pressed); moreover, overall typing speed is inhumanly

```
# Importing the HLISA library
from HLISA.hlisa_action_chains import HLISA_ActionChains

# Creating an ActionChain with HLISA
ac = HLISA_ActionChains(webdriver)

# Selecting an element
element = driver.find_element_by_id('text_area')

# Adding mouse movement and typing with HLISA
ac.move_to_element(element)
ac.send_keys_to_element(element, "Text..")

# Executing a chain
ac.perform()
```

Listing 2: Code example for clicking and sending keys to an element with HLISA.

fast (13,333 characters per minute) and flawless. In our experiments, we observed that fast typing with 10 fingers (600 characters per minute) can cause interleaving key presses, i.e., sometimes a key is only released when a different key has already been pressed.

Moreover, while humans need to press modifier keys to press characters like capital letters, Selenium can input any character that exists without pressing additional modifier keys. By monitoring the usage of modifier keys, detectors can infer the keyboard layout, which can be used for static fingerprinting purposes. In contrast, HLISA ensures dwell time is random, drawn from a normal distribution parametrised with values found in our experiment. In addition, HLISA simulates a key press for the Shift key when needed. Finally, HLISA incorporates contextual pauses based on the measurements due to Alves et al. [1].

Implementation and deployment. To trigger events via HLISA, HLISA functions call the fine-grained interaction functions of the original Selenium API (such as `move_to_offset(x,y)`, `key_down()`, and `key_up()`). This makes HLISA resistant to changes in the Selenium source code that do not affect the Selenium API. The default Selenium API enforces a lower bound on the duration of mouse movements that is too high for simulating human interaction. For Selenium versions <4, we change this duration to 50 msec by overriding the internal Selenium function `create_pointer_move()`. This allows us to express human-like mouse movements.

HLISA's API provides the same calls and signatures as in the original Selenium API (as shown in Table 3 in the Appendix); with the exception of a few additions. This allows developers to integrate HLISA by modifying two lines of code (see red code in Listing 2). Hence, HLISA is compatible with all python projects already using Selenium.

4.2 An arms race model of interaction simulation/detection

Conceptually, websites (as detectors) and web bots (as simulators) are engaged in an arms race. Analogous to the ad blocking arms race as modelled by Storey et al. [32], detectors and simulators can not only refine their current techniques but also escalate the war by introducing stronger techniques.

In Figure 3, we depict a conceptual model of this arms race. Note that other detection mechnisms (e.g., fingerprinting) will give rise to a similar arms race. Simulators begin by exhibiting unlimited

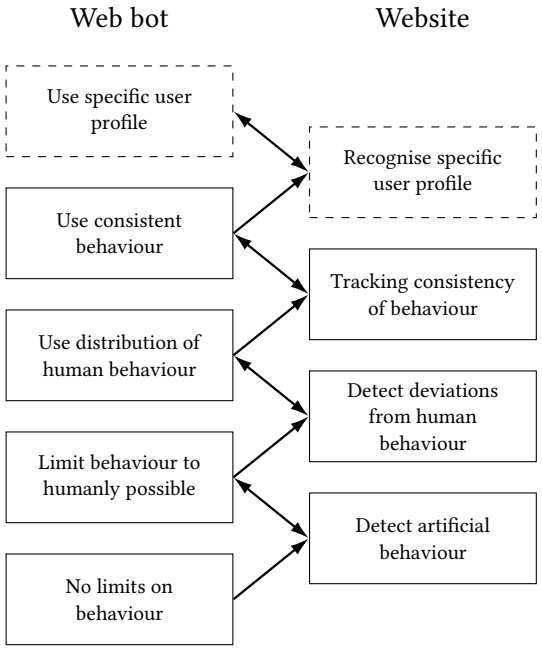

Figure 3: A model of the arms race for page interaction.

behaviour: unhumanly fast, unhumanly perfect, and able to interact with all elements irrespective of visibility. Detectors can identify bots by detecting these aspects. This can cause simulators to limit their interaction to that which is humanly possible: not beyond human speeds, including noise instead of perfect replayability, and accounting for visibility. Either side can refine their techniques further, succeeding in detecting (e.g., detecting artificalness of noise, adding honey elements) or evading detection. In addition, detectors can escalate, by moving from detecting specific aspects of how simulators interact, to detecting deviations from an expected baseline (human interaction). This forces the simulators to move to simulating human interaction. Once again, either side can refine their techniques – in this case, the models on which detection/simulation is based. The next escalation is to recognise that certain interactions are correlated. For example, faster mouse movement may be correlated with higher (or lower) accuracy clicks. Detectors that move to this level will detect simulators that lack such internal consistency in their interactions. Simulators can, of course, adopt such consistency, which will ultimately defeat detection based exclusively on interaction. The detectors can only escalate further by incorporating information beyond interaction (marked with dotted lines in Fig. 3) such as browser fingerprints or individual interaction profiles for specific users (e.g., social media sites have sufficient data for this). This requires an enrolment period during which the detector learns the specific individual's interaction patterns. The only way to defeat such detection mechanisms is to move from simulating interaction that is plausibly human, to simulating the specific interaction profile of a specific individual.

Finally, note that privacy regulations such as EU's GDPR can set a legal limit to how far detectors can evolve. For example, any detection technique that could be used to identify and track a person

would likely fall under the GDPR's purview. The top two detection levels focus detection to such an extent, that individual users could be distinguished. That may run afoul of privacy regulations, though a detailed legal analysis is needed to determine the exact legal limits.

In conclusion, we find that the simulators can always beat the detectors by making use of the same models. This works well for the first few detection strategies, as these are based on generic findings. However, higher up, this becomes more complex: the exact model of consistency needed to satisfy a detector may not be public knowledge. While this complicates matters for the simulators, they have the advantage that detectors must not be too strict or risk barring human visitors entry. Finally, HLISA offers a simulation of human interaction. As such, it is situated at the third level in the hierarchy of Fig. 3. Thus, consistently defeating HLISA requires tracking consistency of behaviour. We caution that HLISA is not the endgame of this level; several refinements are possible to approach human interaction more closely.

5 CONCLUSIONS

In this paper, we (1) investigated side effects of various methods to alter the browser fingerprint, (2) investigated how page interaction using Selenium's interaction API is recognisably different from human interaction, (3) incorporated our findings on interaction into HLISA, a new interaction library for Selenium.

We conclude that fingerprint hiding – in the sense that first-party bot detection can be mostly prevented – is effective. However, we find that spoofing properties in JavaScript can lead to website breakage. This observation, in general is not new, but we were surprised that even simple changes to only the `navigator` object already caused breaks. As such, we advise investigating the compatibility of stealth plugins before using them in large-scale studies. HLISA is publicly available as Python library[1]. Our spoofing extension can be found in the official OpenWPM repository (accepted as draft)[2].

To the best of our knowledge, HLISA is the first comprehensive interaction API that allows Selenium-based bots to hide identifiable behaviour. Based on the conceptual evaluation of an arms race between interaction detectors and simulators, HLISA significantly raises the bar for detectors. Before HLISA, bot interaction was detectable by its artificial nature. To detect HLISA, an interaction-based detector needs to compare the observed interaction to a model of human behaviour. We hope our findings help the research community to improve the reliability of web measurements.

Future work. HLISA's approximation of human behaviour can be further defined. First of all, HLISA's models of human interaction are based on an extremely small set of data. Extending the experiments with more subjects can improve the models underlying HLISA's interaction. Secondly, the interaction of scrolling and mouse movement can be further refined. Mouse movements especially are a rich source for analysis in studies in the human-computer interface field [8, 16, 23]. Related, the simulation of scrolling could be furthered to account for Firefox's `smooth scrolling` setting.

The conceptual discussion of HLISA's limitations offers a framework to reason about its capabilities but lacks concrete data. A practical evaluation would be desirable, but such necessitates detectors. To the best of our knowledge, no study to date has proposed a methodology to identify unknown interaction detectors in the wild with certainty.

Finally, our findings with respect to side effects of fingerprint overriding suggest an investigation comparing stealth extensions is merited. In particular, such an investigation should elucidate how often they cause errors on the visited site and how often they are detected.

[1] https://pypi.org/project/HLISA/
[2] https://github.com/mozilla/OpenWPM/pull/526

REFERENCES

[1] Rui Alves, São Castro, Liliana de Sousa, and Sven Strömqvist. 2007. Influence of typing skill on pause-execution cycles in written composition. *Writing and Cognition: Research and Applications* (01 2007), 55–65. https://doi.org/10.1163/9781849508223_005

[2] Titus Barik, Brent E. Harrison, David L. Roberts, and Xuxian Jiang. 2012. Spatial Game Signatures for Bot Detection in Social Games. In *Proc. 8th Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-12)*. The AAAI Press, 100–105.

[3] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. 2013. Blog or block: Detecting blog bots through behavioral biometrics. *Journal of Computer Networks* 57, 3 (2013), 634–646. https://doi.org/10.1016/j.comnet.2012.10.005

[4] Siebren Cosijn and Nataliya Yasko. 2019. FP-Block 2.0: preventing browser fingerprinting. Bachelor thesis, Open University of the Netherlands.

[5] Luke Crouch. 2018. Data@Mozilla: Improving privacy without breaking the web. https://blog.mozilla.org/data/2018/01/26/improving-privacy-without-breaking-the-web/. last access: October 26, 2021.

[6] Derek Doran and Swapna S. Gokhale. 2011. Web robot detection techniques: overview and limitations. *Journal of Data Mining and Knowledge Discovery* 22, 1-2 (2011), 183–210. https://doi.org/10.1007/s10618-010-0180-z

[7] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. 2015. FPGuard: Detection and Prevention of Browser Fingerprinting. In *Proc. 29th Annual IFIP WG 11.3 Working Conference, DBSec (Lecture Notes in Computer Science, Vol. 9149)*. Springer, 293–308. https://doi.org/10.1007/978-3-319-20810-7_21

[8] Paul M Fitts. 1954. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology* 47, 6 (1954), 381.

[9] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, and Haining Wang. 2009. Battle of Botcraft: fighting bots in online games with human observational proofs. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS'09)*. ACM, 256–268. https://doi.org/10.1145/1653662.1653694

[10] Cristiano Giuffrida, Kamil Majdanik, Mauro Conti, and Herbert Bos. 2014. I Sensed It Was You: Authenticating Mobile Users with Sensor-Enhanced Keystroke Dynamics. In *Proc. 11th Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'14) (Lecture Notes in Computer Science, Vol. 8550)*. Springer, 92–111. https://doi.org/10.1007/978-3-319-08509-8_6

[11] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean Michel Picod, and Elie Bursztein. 2016. Cloak of Visibility: Detecting When Machines Browse a Different Web. In *Proc. 37th IEEE Symposium on Security and Privacy (S&P'16)*. 743–758. https://doi.org/10.1109/SP.2016.50

[12] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. 2019. Fingerprint surface-based detection of web bot detectors. In *Proc. 24th European Symposium on Research in Computer Security (ESORICS'19) (LNCS)*. Springer, 586–605. https://doi.org/10.1007/978-3-030-29962-0_28

[13] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. 2021. Towards Realistic and Reproducible Web Crawl Measurements. In *Proc. 31st The Web Conference (WWW'21)*. ACM, 80–91. https://doi.org/10.1145/3442381.3450050

[14] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. 2017. FPRandom: Randomizing Core Browser Objects to Break Advanced Device Fingerprinting Techniques. In *Proc. 9th Symposium of the Engineering Secure Software and Systems (ESSoS'17) (Lecture Notes in Computer Science, Vol. 10379)*. Springer, 97–114. https://doi.org/10.1007/978-3-319-62105-0_7

[15] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proc. 26th Network and Distributed System Security Symposium (NDSS'19)*. The Internet Society, 1–15. https://doi.org/10.14722/ndss.2019.23386

[16] Byungjoo Lee and Hyunwoo Bang. 2013. A kinematic analysis of directional effects on mouse control. *Journal of Ergonomics* 56 (09 2013). https://doi.org/10.1080/00140139.2013.835074

[17] Linder. 2020. User-Agent Switcher. https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher-revived/. last access: October 26, 2021.
[18] neroux. 2020. Random User-Agent. https://web.archive.org/web/20210303001755/https://addons.mozilla.org/en-US/firefox/addon/random_user_agent/. last access: October 26, 2021.
[19] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. 2015. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *Proc. 24th International Conference on World Wide Web (WWW'15)*. ACM, 820–830. https://doi.org/10.1145/2736277.2741090
[20] Robbert Noordzij. 2019. Synthetic Fragmentation Experiments using WildFragSim. Bachelor thesis, Open University of the Netherlands.
[21] White Ops. 2016. The Methbot Operation. https://www.whiteops.com/hubfs/Resources/WO_Methbot_Operation_WP.pdf. last access: October 26, 2021.
[22] KyoungSoo Park, Vivek S. Pai, Kang-Won Lee, and Seraphin B. Calo. 2006. Securing Web Service by Automatic Robot Detection. In *Proc. USENIX Annual Technical Conference (ATC'06)*. 255–260.
[23] James Phillips and Thomas Triggs. 2001. Characteristics of cursor trajectories controlled by the computer mouse. *Journal of Ergonomics* 44 (05 2001), 527–36. https://doi.org/10.1080/00140130121560
[24] Nataasha Raul, Radha Shankarmani, and Padmaja Joshi. 2020. A Comprehensive Review of Keystroke Dynamics-Based Authentication Mechanism. In *Proc. International Conference on Innovative Computing and Communications (ICICC 2019)*. Springer, Singapore, 149–162. https://doi.org/10.1007/978-981-15-0324-5_13
[25] Ray. 2020. User-Agent Switcher and Manager. https://addons.mozilla.org/en-US/firefox/addon/user-agent-string-switcher/. last access: October 26, 2021.
[26] Alexander Schlarb. 2020. User-Agent Switcher. https://addons.mozilla.org/en-US/firefox/addon/uaswitcher/. last access: October 26, 2021.
[27] Michael Schwarz, Florian Lackner, and Daniel Gruss. 2019. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Proc. 26th Annual Network and Distributed System Security Symposium (NDSS'19)*. The Internet Society. https://doi.org/10.14722/ndss.2019.23155
[28] sereneblue. 2020. Chameleon. https://addons.mozilla.org/en-US/firefox/addon/chameleon-ext/. last access: October 26, 2021.
[29] Abdul Serwadda and Vir V. Phoha. 2013. Examining a Large Keystroke Biometrics Dataset for Statistical-Attack Openings. *Journla of ACM Transactions on Information and System Security* 16, 2 (2013), 8:1–8:30. https://doi.org/10.1145/2516960
[30] Abdul Serwadda, Vir V. Phoha, Zibo Wang, Rajesh Kumar, and Diksha Shukla. 2016. Toward Robotic Robbery on the Touch Screen. *Journal of ACM Transactions on Information and System Security* 18, 4 (2016), 14:1–14:25. https://doi.org/10.1145/2898353
[31] Sergey Shekyan. 2015. Detecting PhantomJS based visitors. https://blog.shapesecurity.com/2015/01/22/detecting-phantomjs-based-visitors/. last access: October 26, 2021.
[32] Grant Storey, Dillon Reisman, Jonathan R. Mayer, and Arvind Narayanan. 2017. The Future of Ad Blocking: An Analytical Framework and New Techniques. *CoRR* abs/1705.08568 (2017). http://arxiv.org/abs/1705.08568
[33] Pang-Ning Tan and Vipin Kumar. 2002. Discovery of Web Robot Sessions Based on their Navigational Patterns. *Journal of Data Mining and Knowledge Discovery* 6, 1 (2002), 9–35. https://doi.org/10.1023/A:1013228602957
[34] Christof Ferreira Torres, Hugo L. Jonker, and Sjouke Mauw. 2015. FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting. In *Proc. of the 20th European Symposium on Research in Computer Security (ESORICS'15), Proceedings, Part II (LNCS, Vol. 9327)*. Springer, 3–19. https://doi.org/10.1007/978-3-319-24177-7_1
[35] Antoine Vastel. 2018. Detecting Chrome headless, new techniques. https://antoinevastel.com/botdetection/2018/01/17/detect-chrome-headless-v2.html. last access: October 26, 2021.
[36] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Xavier Blanc. 2020. FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers. In *Proc. 2nd NDSS Workshop on Measurements, Attacks, and Defenses for the Web (MADWEB'20)*. 2–14. https://doi.org/10.14722/madweb.2020.23010
[37] W3C. 2020. WebDriver. https://www.w3.org/TR/webdriver/. last access: October 26, 2021.
[38] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollsén, and Martin Lopatka. 2020. The Representativeness of Automated Web Crawls as a Surrogate for Human Browsing. In *Proc. 21st The Web Conference 2020 (WWW '20)*. Association for Computing Machinery, 167–178. https://doi.org/10.1145/3366423.3380104

A ETHICS

Involvement of human participants

The experiments in this paper involving human subjects were extremely limited in scope. The goal of these experiments was not to establish an average of generic human behaviour, but to contrast Selenium's interaction with that of an individual human. As such, we limited the involved human participants to ourselves. Our institution has an ethical advisory board, separate and independent from its IRB, as the formal IRB process is typically very long, while the advisory board can give advice within a few workdays. We asked the ethical advisory board for advice on whether approval by the IRB was necessary. The advisory board agreed that in this case, for the specific, limited amount of data gathered from the authors themselves, a full, formal ethical review process was not needed.

Unintended negative secondary effects (dual use)

Though our work aims to improve the accuracy of tooling used in scientific studies of the web, we realise that our work may have secondary effects. The goal of our work is to make web bots less distinguishable from regular visitors. This may be leveraged by nefarious bots as well as benign bots.

In fact, malicious web bot campaigns already incorporate human simulated interaction that circumvents fraud detection (for example, Methbot [21]). Methbot's interaction capabilities seem to be at least as good as those of HLISA, so HLISA is not extending the capabilities of criminals. This leaves benign uses of HLISA: for researchers, to augment their scraping bots; and for websites, to augment their bot defences (by incorporating behavioural bot detection trained against HLISA).

With these effects in mind, we believe the beneficial effects to outweigh the potential negative consequences of our research.

Limited applicability of the model

HLISA's interaction model is based on the data collected from white, male, Western-European, highly educated subjects studying computer science. These subjects are obviously not representative of the world's population. As such, we caution against using HLISA as-is for other purposes (e.g., usability testing).

B ERROR OCCURRENCE WITHIN HTTP TRAFFIC

Figure 4 shows the occurrence of error-related HTTP responses from our evaluation in Section 3.2. In general, OpenWPM does not retrieve a far larger number of error responses, when being

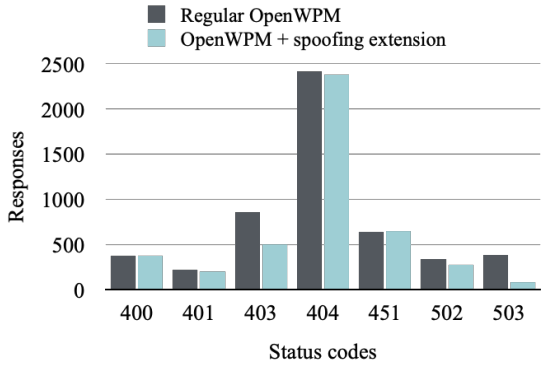

Figure 4: HTTP (error) responses listed by status code with more than 100 occurrences.

detectable. However, we see a significant variation for responses with 403 and 503 codes, which refers to blocking.

C EVENTS RELATED TO OR TRIGGERED BY INTERACTION

- **Document:**
  - copy
  - cut
  - dragend
  - dragenter
  - dragleave
  - dragover
  - dragstart
  - drag
  - drop
  - fullscreenchange
  - gotpointercapture
  - keydown
  - keypress
  - keyup
  - lostpointercapture
  - paste
  - pointercancel
  - pointerdown
  - pointerenter
  - pointerleave
  - pointermove
  - pointerout
  - pointerover
  - pointerup
  - scroll
  - selectionchange
  - selectstart
  - touchcancel
  - touchend
  - touchmove
  - touchstart
  - transitionend
  - transitionrun
  - transitionstart
  - visibilitychange
  - wheel
- **Element:**
  - auxclick
  - blur
  - click
  - contextmenu
  - dblclick
  - focusin
  - focusout
  - focus
  - mousedown
  - mouseenter
  - mouseleave
  - mousemove
  - mouseout
  - mouseover
  - mouseup
  - select
- **Window:**
  - resize
  - focus

D MEASURING SELENIUM'S INTERACTION

To analyse the behaviour exhibited by Selenium interaction API, we first determined how to measure any interaction. OpenWPM uses the Firefox browser, which offers 57 events (see Appendix C) related to or triggered by interaction. Many of these provide overlapping information. The following set of 10 events together cover all interaction information available to a web page:

- **Mouse movements:**
  - mousemove
- **Mouse clicking:**
  - dblclick
  - mousedown
  - mouseup
- **Scrolling:**
  - scroll
  - wheel
- **Typing:**
  - keydown
  - keyup
- **Touch:**
  - touchstart
  - touchend
- **Losing/gaining focus:**
  - visibilitychange
  - blur
  - focus

Interestingly, the granularity of events varies. In particular, the granularity of **mouse movement** events can vary. Moreover, we did not find a correlation between the number of events fired per second and mouse movement speed. All in all, we found Firefox's event API too coarse to register every detail of normal mouse movement. In contrast, events for all other categories provide a more complete view. For example, the granularity for **typing** events is 1ms. Interestingly, for **mouse clicking**, Firefox asks its environment what the maximum interval is to consider two consecutive clicks a double click. For Windows, this defaults to 500ms; in testing Selenium, we found a maximum interval of 600ms.

Scroll-related events can be triggered in many ways, including: mouse wheel, trackpad scrolling, scroll bar, arrow keys, using find, URL anchors, auto scrolling. This wide array of origins, each causing different amount of scrolling, significantly restricts the effectiveness of using this type of interaction to distinguish bots from human visitors. Thus, even though the amount scrolled by a scroll-wheel 'click' is fixed (57 pixels in our setup), different scrolling amounts can and will occur in normal use. Absence of a `wheel` event, or different amount of scroll distance thus do not suffice to distinguish a web bot from humans. **Touch** movement can also indicate a touchpad, and, as such, is included in `mousemove`. Finally, **focus** events can be triggered by Selenium also after minimising a headful browser. Minimising causes a `visibilitychange` event, after which no further interaction should occur. This should be addressed in experiment design.

E MEASURING HUMAN INTERACTION

We built a website that uses JavaScript to record events. For each interaction type, our site asks the user to perform simple tasks.

For mouse movement, we record the cursor coordinates and timestamps for each `onmousemove` event. We use this data to monitor the distance, slope and speed of the cursor as well as to create a visual representation of the taken path. The site instructed the participant to click two distant elements in a specific order, so that the interaction starts and ends at similar positions.

For mouse clicks, we created a moving element to collect data for various different angles. The element relocates every time after it is clicked. Our human participant repeated this task 100 times, where we recorded the dwell time and the position of each click.

For scrolling events, we created a page with a sufficient height (30K pixels). The task was to scroll via the mouse wheel from top to bottom at a comfortable pace. This provides realistic data on one scrolling method; other methods are not considered.

Last, we took measurements on typing by letting user type a given text of 100 characters. Our page recorded key press and key release events, including the timestamp when they occurred. From this, we derived dwell and flight times for a user's keystrokes. (Dwell time denotes the time between a key press and a key release; flight time is the time between a key release and key press.) We combined our data with the timing of various pauses from the work of Alves et al. [1]. Our distribution thereby takes into account a variety of timings of events, such as pauses after opening or closing a sentence, writing a new word, using commas, and many more.

Table 3: The HLISA API.

API function	Arguments	Description
HLISA_ActionChains()	webdriver	Constructor to create an action chain
perform()		Executes actions in a chain
reset_actions()		Removes all actions from the current chain
pause()	duration	Pauses the execution of the action chain (in sec)
move_to()	x,y	Moves the cursor from the current position to a given position
move_by_offset()	x, y	Moves the cursor relative to the current position
move_to_element()	element	Moves the cursor to a position within an element's boundaries
move_to_element_with_offset()	element, x, y	Moves the cursor relative to an element's top-left corner
move_to_element_outside_viewport()	element	Scrolls element into the viewport before using `move_to_element`
click()	element	Clicks. If element is provided, first performs `move_to_element`
click_and_hold()	element	Same as `click` without release action
release()	element	Same as `click` without press action
double_click()	element	Same as `click` with an additional click shortly after the first
send_keys()	keys	Executes a human typing rhythm for the given keys
send_keys_to_element()	element, keys	Selects the element, then executes the `send_keys` function
scroll_by()	x, y	Scrolls the viewport till a distance is taken
scroll_to()	x, y	Scrolls until the specified position is in the top left corner
context_click()	element	Same as `click` using a right mouse button
drag_and_drop()	element1, element2	Press left button over element1, move mouse to element2, release mouse button
drag_and_drop_by_offset()	element, x, y	Press left mouse button on element, moves to target offset (x, y) and releases button

functionname() replacement for Selenium function to interact human-like
functionname() new function, not available in Selenium
functionname() passthrough to Selenium's implementation

F CURRENT LIMITATIONS OF HLISA

The current implementation of HLISA is limited in its capability to impersonate human interaction in two general ways. First, there are some key aspects of human behaviour that transcend individual interaction with a page, but concern more generic behavioural aspects which need to be seperately modelled. Whether and to what extent such behaviour should be simulated depends on the specific experiment being conducted and thus should not be integrated into HLISA. For example, human visitors may exhibit non-functional interaction with webpages, such as selecting and deselecting parts of a page without purpose. Such idiosyncrasies of human behaviour cannot be executed independently by an interaction API, as it may interfere with an experiment's purpose. In a similar vein, there are other aspects of interaction that increase distinctiveness, which should be handled on the level of an experiment, outside of any specific interaction API. These include:

- Mouse movement starting at (0,0), which can be solved by moving the mouse prior to loading a page
- Adding random/spontaneous mouse movements
- Misclicking
- Introducing typing errors and more complex typing behaviour such as reformulating sentences, pausing in longer texts, erasing and cancelling input.

While such aspects cannot be delegated to an interaction API, there are several ways to further refine HLISA to better approximate human interaction. A general caveat is that HLISA currently uses a normal distribution (parameters following from our experiments) to introduce noise in behaviour, while human behaviour is not normally distributed [3]. Similarly, HLISA is a proof-of-concept interaction API. It accounts for basic measurements of interaction, such as dwell and flight times of clicks and key presses. Advanced measures of interaction, such as adapting mouse movement to target size and shape, go beyond its proof-of-concept nature. For the same reason, HLISA does not account for touch actions.

G HLISA IN COMPARISON TO OTHER TOOLS

In Table 4, we compare the features covered by HLISA with other tools that simulate parts of human interaction. These come from a variety of sources,ranging from tools focusing on browser interaction (e.g., Scroller), to tools focusing on scripting games (e.g., BezMouse).

Table 4: A comparison of different libraries or code samples to simulate human like behaviour. A '✓' indicates the functionality is present in the library or code sample

Functionality	Package							
	HMM[1]	**PyC**[2]	**BezMouse**[3]	**pyHM**[4]	**Scroller**[5]	**ClickBot**[6]	**[20]**[7]	**HLISA**
Mouse movement functionality	✓	✓	✓	✓		✓	✓	✓
Realistic mouse movement speed		✓	✓	✓		✓		✓
Movement accelerates/decellerates		✓	✓	✓				✓
Movement shivering		✓		?[a]				✓
Curve in movement[b,c]	✓	✓	✓	?[a]		✓		✓
Moves to random location in element[b]								✓
Click functionality		✓	✓	✓		✓	✓	✓
Realistic dwell time[b]				?[a]		✓		✓
Simulates accidental right click			✓					
Simulates accidental double click			✓					
Simulates accidental no click			✓					
Scrolling functionality					✓			✓
Pause between scroll ticks					✓			✓
Pause for finger replacement					✓			✓
Realistic scroll distance in tick					✓			✓
Keyboard functionality							✓	✓
Flight time[b]							✓	✓
Dwell time[b]								✓
Timings based on data							✓	✓
Other features								
Selenium ready	✓				✓			✓

1. "Human-like mouse movement": answer using B-spline curves to question on StackOverflow https://stackoverflow.com/a/48690652
2. Pyclick: Python library for mouse movement using Bézier curves https://github.com/patrikoss/pyclick
3. BezMouse: Python tool for mouse movement using Bézier curves, to avoid bot detection in games https://github.com/vincentbavitz/bezmouse
4. Python Human Movements: Python package to simulate human movement https://pypi.org/project/pyHM/
5. Scroller: tool to simulate human scrolling in Selenium https://github.com/hayj/Scroller
6. ClickBot: Java tool to simulate mouse movement and clicks https://github.com/amSangi/ClickBot/
7. Bachelor thesis, incorporates typing rhythm from HCI literature in Java framework

a. The project links to source code that is incomplete
b. Absence of this feature makes interaction obviously artificial
c. Previously required to bypass Google reCaptcha https://stackoverflow.com/a/37220168