Research Internship

Camouflaging OpenWPM

Author:
David Roefs
s4666623

Supervisors:
dr. ir. Hugo Jonker
Benjamin Krumnow, MSc

Reviewer:
dr. ir. Erik Poll

Presentation date:
11-06-2021

June 2021

Abstract

Web bot detection followed by web page alteration might influence results of large-scale research on the web. While research has been done to mitigate this threat in respect to web bot fingerprinting, interaction based web bot detection has not yet been thoroughly investigated in the literature. In this work, an extensive analysis is performed in order to identify means by which interaction can be observed by web sites. This is followed by the implementation of HLISA, a new API that replaces the Selenium interaction API in order to make web bots that interact with web pages less detectable.

Contents

Chapter 1

Introduction

Websites do not treat all visitors in an equal manner. One factor influencing treatment is whether the visitor is a human or a web bot. Content that is expensive to serve, like images or video, are better preserved for the intended audience. Serving adverts to bots is a lost cause, and so is tracking bots with the intent of creating a detailed profile. Considering 50% of web traffic consists of web bots [RD15], it makes sense for web site owners to distinguish between bots and regular users. Technology for this is already available, as web bot detection is its own field and industry [ASLN20].

While there are valid and important use cases for distinguishing web bots from regular users, this distinction may pose a problem for large-scale research on the web, since this depends on the use of web bots. For example, OpenWPM is a web bot built for privacy measurements of websites [EN16]. It has been used in over 70 studies[1]. If sites serve OpenWPM different content than humans, results of the studies might be incorrect. That this threat is not just theoretical, is supported by findings from recent studies. Englehardt and Narayanan concluded that PhantomJS, a different web bot, receives less advertisements [EN16]. Jonker et al. [JKV19] found differences between responses to web bots and responses to human visitors, and differences in responses to different web bots. Goßen [Goß20] found sites that reacted differently to OpenWPM after the removal of a property identifying OpenWPM as a web bot. Finally, the results of Krumnow et al. [KJK21] indicate OpenWPM being detected on ∼12% of the 100,000 top websites of the Tranco list. Sometimes, detection is specifically tailored to detect OpenWPM.

Krumnow et al. investigate the effect of making OpenWPM less detectable [KJK21] on web site responses. They found, amongst other, that 14% of sites detect bots on the front page, a number which increases to 19% if subpages are subsequently visited. Jansen [Jan21] provides an initial investigation of the prevalence of web bot detection based on the interaction patterns of a visitors. This interaction based bot detection turned out to be more difficult to identify than browser fingerprinting-based bot detection: access to fingerprint attributes unique to web bots is clear proof of bot detection, while there is no such clear delineation for interaction based detection.

Contributions. The main contributions of this work are:

- An extensive evaluation of how browser events enable interaction based detection,
- HLISA, a Python package that replaces the Selenium ActionChains API with an API that makes interaction based detection harder.

Methodology. The current work focuses on mitigating interaction based detection of OpenWPM. To achieve this, first an analysis will be performed to identify all means by which interaction can be detected. Then a Python package will be created that can be used in combination with OpenWPM and limits the difference between interaction of OpenWPM and humans. Methodology of each of these steps is further detailed in their chapters.

[1] `https://webtap.princeton.edu/software/`, retrieved 2-04-2021

Chapter 2

Related work

This work builds directly on other work in the field of web bot detection mitigation, but also on other categories of research: research into human-computer interaction and research in detecting web bots by their interaction patterns. The latter two categories will be discussed first.

Real human interaction. As a major goal of this work is to mimic human interactions, it is apparent that knowledge of how humans interact should be incorporated in this work. The most prominent model of human interaction in the field of human-computer interaction is undoubtedly Fitts' law [Fit54], the famous model for the difficulty of reaching a target by a means of user input, $\textit{difficulty} = \log_2(\frac{2l}{s})$ with l the length of the movement and s the target size. Fitts' law can be used to verify models of human interaction and as a guideline when creating new models when movement time, distance and accuracy are involved. The law is shown to hold over a wide range of conditions [PA97] and for tasks like mouse movements [SM04] and scrolling [HCBM02].

Considering mouse movements, many studies into different aspects of the movement have been performed. Grahma et al. [GM96] found that the idea of a pointing movement consisting of two phases, introduced by Woodworth [Woo99], also holds for digital pointing. Phillips et al. [PT01] studied mouse movement trajectories, contributing concrete statistics on aspects like overshoot distances and angles of approach. Angle of approach is a factor in the duration of a movement as Whisenand et al. [WE96] and Lee et al. [LB13] showed.

Noordzij [Noo19] simulated human typing interaction to investigate fragmentation of files on a hard disk. The simulation needed to approach human interaction properties to such a degree that functions like autosaving by a text editor or deleting temporary files by the operating system happen on moments comparable to a situation with a human typist. A difference with the current work is the observing party. The simulation in Noordzij's thesis is not aimed at observers actively trying to discriminate between a human and a simulator, and can therefore be less detailed. Such details have been subject of research by Alves et al. [ACdSS07], who compared slow and fast typists in order to find how important automation is in the task of typing. Along with their findings, they published detailed statistics on typing patterns. These statistics are valuable for both bot detection and bot detection mitigation.

Detecting bots by interaction properties. In order to mitigate interaction based detection, it is important to know how bots can be detected via interaction. Barik et al. [BHRJ12] show how bot detection by mouse movements can be used to detect bots in games that run in a web browser. They implemented bots with different kinds of interaction properties which are also exhibited by real bots, contributing a classification of bots too. They show also advanced bots can be recognized. Gianvecchio et al. [GWXW09] also classify bots in online games and show bots have limited variety in their interaction patterns compared to humans. Chu et

al. [CGK+13] introduce interaction based detection to websites in order to block bots, utilizing cursor movements and keystrokes.

It is also possible to detect individual users by their interaction properties. Krátky et al. [KC18] created a system that correctly identifies a user from a group of 100, with a 95% accuracy. Increasing the group size to 2000 reduces accuracy to 80%. Earlier, Ahmed and Traore [AT07] already concluded mouse movements can be used to recognize users to enhance security of applications, building on other user identification methods based on mouse movement or typing dynamics. This shows that even mimicking human interaction perfectly might not be enough to prevent being identified as a web bot.

Jansen [Jan21] created an overview of methods for interaction based detection. After manually investigating web bot detection scripts and creating a framework to automatically recognize interaction based detection using those findings, Jansen showed interaction based detection is being used in practice on the internet, indicating the need for web bot detection mitigation.

Bot detection mitigation. Vlot [Vlo18] and Jonker et al. [JKV19] identified properties that facilitate fingerprinting web bots. Goßen [Goß20] builds on this work by analyzing the stability of OpenWPM's detectable properties, investigating measures to counter fingerprinting those properties and ultimately creating a plugin that removes one of the most prominent properties from OpenWPM's fingerprint. This is a first step into creating a version of OpenWPM that is harder to detect. Krumnow et al. [KJK21] identified more properties that allow detecting OpenWPM, including categories of interactional properties, as part of a broader investigation. These identified categories of interactional properties are: typing, scrolling, mouse clicking and mouse movement. The properties have been inferred by analyzing the Selenium API. The current work analyses all events that can be triggered in browsers in an attempt to find even more detectable interactional properties. The combination of all found properties are used in the current work to create a library that limits the interactional properties web bot detectors can observe from OpenWPM, as was left as future work by Krumnow et al.

Chapter 3

Analyzing interactions

Web bot detectors analyze properties of visitors to discriminate between web bots and humans. The presence or absence of a single property, like a specific user agent string or the `webdriver` attribute can be enough to conclude a visitor is a web bot. In case no single property distinguishes the web bot from humans, combinations of properties (also known as a *browser fingerprint*) may still suffice to make this distinction [JKV19]. A specific combination of properties can reveal whether the visitor is a web bot or a human.

Another approach to detect web bots is to analyze the interactions of the visitor on the web page. Interactions, like moving the mouse or typing on the keyboard, reveal large amounts of information about the visitor performing the actions. Whether interaction on the web page by the visitor is present is in itself a property of the visitor, as web bots may not perform interactions at all. As such, the absence of interactions alone can, and has been used, as a web bot detection method [PPLC06].

In this work however, the way of interaction itself is the sole subject of investigation. It is assumed that interaction is present, and the question is whether this interaction stems from a human or from a web bot. In contrast to the methods discussed earlier, the task is not to detect the presence of a (set of) properties, but to classify the observed interactions in either "interactions produced by a human" or "interactions produced by a web bot".

This chapter consists of an analysis of how the task of classification can be performed in the context of who will be performing this classification. Therefore, after setting the scope of the analysis, a state model is presented which divides both web bot operators and web site operators in groups depending on the techniques they use. Then, an extensive summary will be made of interactional properties that can be used in the classification task. This is followed by a comparison of human and web bot interactions, which shows how one can differentiate between humans and web bots.

3.1 Scope & limitations

It has already been stated that in this analysis, only the way of interacting – not the presence or absence of interaction – is considered. Interactions on a website can be divided in two categories:

- **intra-page interactions:** interactions within a page, e.g. duration of key presses and mouse clicks, mouse movement, etc.
- **inter-page interactions or behavior:** interactions over multiple pages, e.g., time spent on one page, number of pages visited, path over the site, etc.

Inter-page behavior has been subject of many studies [WLS19], but is not subject of this analysis.

Intra-page interaction can be divided into two sub-categories: *actions*, and *chains of actions*. *Actions* are expressions of intra-page interaction, like moving the mouse or typing on a keyboard. *Chains of actions* are sequences of *actions*, including timings between *actions* and other events. A *chain of actions* can consist of a single *action*. Such a *chain of actions* still differences from an *action*, as the *action* stands on its own, while the *chain of actions* with one element also considers the context of that action.

In Figure 3.1 two cursor trajectories are visualized. Both trajectories start in point A and end in point C, both going through point B. The difference between both trajectories is that the gray trajectory is a chain of two actions: a movement from A to B, and a movement from B to C. The black trajectory is a chain of one action: a movement from A to C.

Clearly, initiating two movements instead of one has consequences for the observed cursor movement. This also follows from Fitts' law: $2 \times \log_2(\frac{2l}{s}) \neq \log_2(\frac{2\times 2l}{s})$, with l being the length from one point to the next, and s being the size of the point. This difference between the gray and black cursor move *actions* can be accounted for by both web bot operators and web site operators, as l and s are properties of the web page the movement is performed at.

What cannot be accounted for is the context of the *actions*, which is present in *chains of actions*. The duration of the pause between the two cursor movements in the gray trajectory depend on the semantics of the web page, and the intention of the user. These variables are unknown. Therefore, only *actions* are considered in this analysis, while *chains of actions* are left for future work.

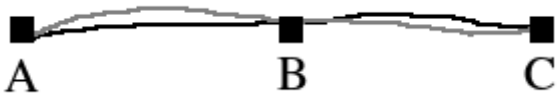

Figure 3.1: Two separate cursor moves in gray and one cursor move in black.

Finally *actions* can be divided into atomic event actions and composite actions as defined by Krumnow et al. [KJK21]. Both atomic and composite actions are part of the analysis. The scope is visualized in figure 3.2.

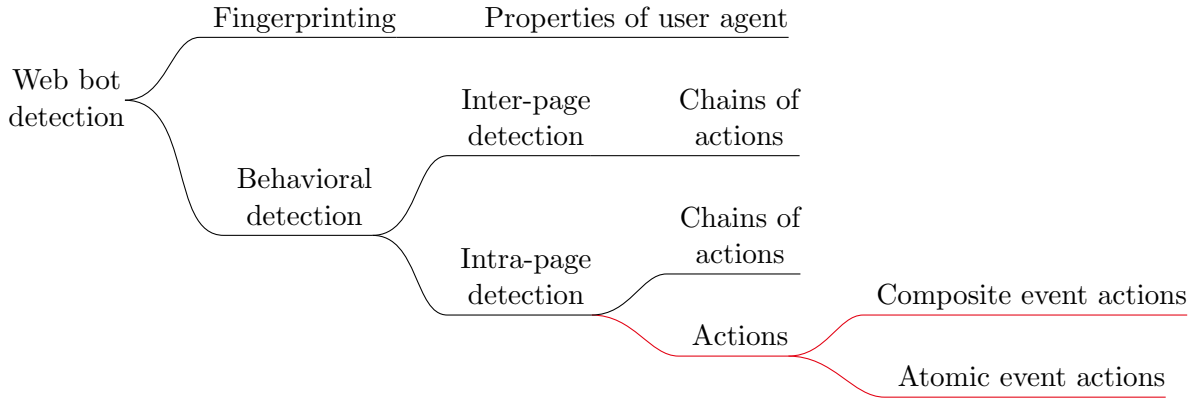

Figure 3.2: Taxonomy of web bot detection. Red edges indicate the current scope.

3.2 State model

As stated in the introduction of this chapter, bot detection based on interaction properties is a classification problem. Both a human and a web bot produce interaction properties and it is up to the website to discriminate between these two kinds of visitors based on the observed interaction properties. Thus, not only what can be observed is important to analyze, but also the party that observes the interactions is a factor which influences the result of classification.

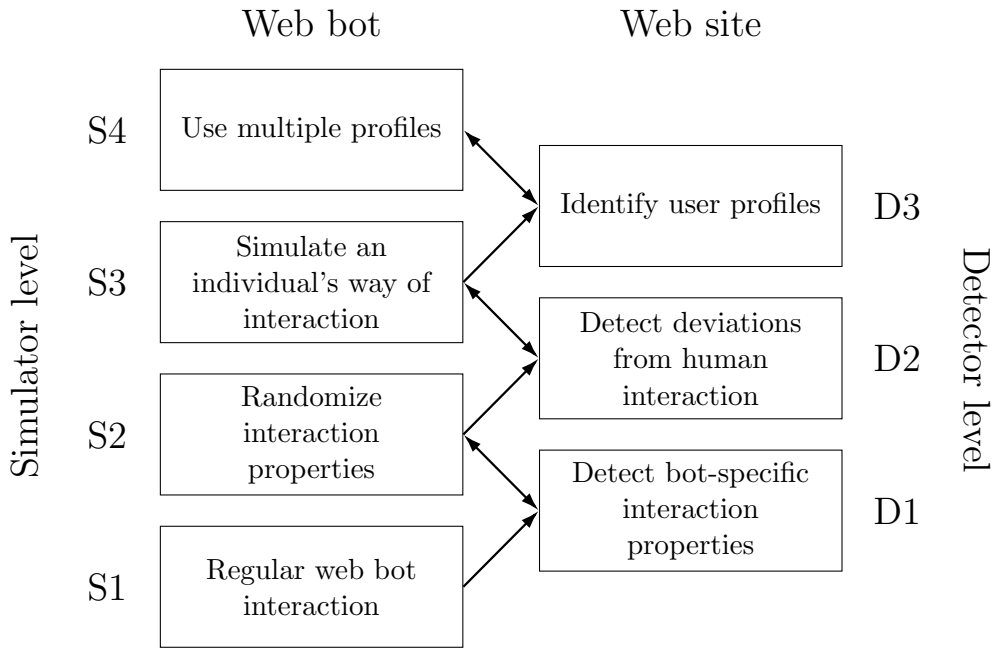

Figure 3.3: A model of the detection/simulation arms race for page interaction.

In this analysis, the observing party – the owner of the web page – is called the *detector* and the opposing party, the party operating the web bot, is called the *simulator*.

Both the detector and the simulator can be in a certain state. The state space can be seen as an "arms race" between both parties. If one party takes a significant step in techniques used, it transitions to the next state level. This implies the party has a large advantage over the other party. This model is inspired by the state diagram of Storey et al. [SRMN17] in their article about ad blocking. An important difference is that in the interaction based detection model, an arms race exists for every category of interaction. The detector can defeat the simulator by using a new category of interaction in state D1, even if the simulator is in state S4 in other categories of the interaction based detection model.

In the model, the arms race starts when detectors start to detect bot-specific interaction properties. The simulator, in order to prevent being detected by the bot-specific interaction properties it elicits, has to react by making the interaction properties more random. The detector can try to also recognize these randomized interaction properties, creating a mini-arms race between state S2 and D1.

To end this min-arms race, the detector can turn the approach around. Instead of only detecting interaction properties specific to bots, it detects interaction properties that deviate from human behavior. Now, the simulator also has to raise the bar and starts to simulate human like interaction.

Faced with human-like interaction, the detector can escalate one last time. Now, the detector starts to recognize the individual interaction properties of visitors. Visitors that elicit interaction properties attributed to a known bot are classified as bots. The simulators can counter this by simulating new and unique visitors, that are not yet classified as a bot. In a last attempt, the detector can start to use lists of interaction properties that are known to belong to a human by using third party services.

But tracking individual users is already subject of interest of privacy activists[2], browser vendors[3] and academic research [EN16]. Identifying individual visitors might also be illegal [Bor16]. If anonymity on the web stays preserved, the simulators will ultimately win the

[2] `https://privacybadger.org/`

[3] `https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop`

arms race – the detector will only see yet another visitor that interacts like a human.

In practice, lots of detectors and simulators exist which are all on different levels in the state space. For an analysis of such simulators, see Appendix C.

3.3 What the detector can observe

OpenWPM, the web bot subject of this analysis, uses Firefox in combination with Selenium and custom tooling to crawl. Using Firefox helps to get results that are representative. By using Firefox, the page is loaded and rendered in a completely standard way. This is also an important advantage to remain undetected. Users that do not enable JavaScript are seen as suspicious [CGK+13], as bots often have JavaScript disabled. The large disadvantage of enabled JavaScript is that it introduces a rich set of events for the detector to monitor interaction of a user. Intra-page interactions can only be observed by the detector through these events.

Therefore, in the rest of this chapter, all existing events will be analyzed to see which kind of interaction properties they enable the detector to observe, and how this can be used to detect web bots. As implementation specific details of events differ for different browsers, Firefox will be used as reference when it comes to specific details of events, as Firefox is currently the only browser supported by OpenWPM.

3.3.1 Interaction properties exposed

Firefox as used in OpenWPM exposes interaction properties via events. A listing of events is given in the Event reference[4] of the MDN Web Docs. This incomplete[5] list contains 356 events. They are analyzed to determine which intra-page interaction properties can be observed by the detector. Although the complete list is analyzed, interaction might be observed in unexpected ways by using (combinations of) events in specific ways. As will be seen in later chapters, some methods to observe interaction are not apparent. Therefore, the findings in this chapter may be incomplete.

A first selection of events is performed by only selecting events that reveal any properties of interaction to the detector. This selection is done by evaluating all events based on their category, name, and in case of doubt their documentation. Touch actions are not selected, even though they let the detector observe touch interactions. Selenium implements a touch API[6] but using touch actions will make the fingerprint more unique, therefore, those are not recommended to use. When touch actions are not used, touch events will also reveal no information to the detector.

After the first selection, 57 events remain. These events are listed in Appendix B. Many of the selected events are clearly related to interaction, like mouse move events and key press events. Other events may seem unrelated to interaction at first. For example, the event `transitionstart` is included, which indicates a CSS transition started. This event is included because information can leak indirectly via this event. A CSS transition can be started when the mouse hovers over the element, at which point the `transitionstart` event is fired. The detector can place (hidden) elements on the page, and from the order and timing of the events infer the movements of the mouse.

The selection can be further reduced by removing events that provide the same information as other events. For example, after a more thorough investigation, the `transitionstart` event is removed as it provides no additional information over the `mousemove` event. The second selection brings the list down to 10 events. Grouped into the four categories found

[4] `https://developer.mozilla.org/en-US/docs/Web/Events`

[5] On the page it is stated: "This page lists many of the most common events you'll come across on the web"

[6] `https://www.selenium.dev/selenium/docs/api/py/webdriver/selenium.webdriver.common.touch_actions.html#module-selenium.webdriver.common.touch_actions`

by Krumnow et al. [KJK21] and one additional one, the events that leak unique properties of *action*-based interaction are:

- **Mouse movements:**
  - `mousemove`
- **Mouse clicking:**
  - `dblclick`
  - `mousedown`
  - `mouseup`
- **Scrolling:**
  - `scroll`
  - `wheel`
- **Typing:**
  - `keydown`
  - `keyup`
- **Leaving and entering:**
  - `visibilitychange`
  - `blur`

The remaining events will now be discussed per category.

3.3.2 Mouse movements

When the cursor is moved from position a to position b, the `mousemove` event is fired. The event produces data, most noticeably the new x and y position of the cursor. In theory, this event would fire every time the cursor moves at least one pixel. In practice when constantly moving the cursor on a high speed from one side of the screen to the other, only once every 17 milliseconds an event is fired on average.

How detailed the detector can observe the cursor movement depends on the cursor movement speed. If a user takes 30 seconds to cross a 1920x1080 pixel screen at constant speed, approximately every moved pixel receives its own event. In this case, the detector has a detailed view of the *cursor trajectory*. If it takes the user only 3 seconds to cross the screen, 90% of the pixel locations are "lost" as no event is fired at those locations. The detector now will have a general idea of how the trajectory looked. If the cursor crosses the screen in 300 milliseconds, the detector is left with only 1% of the pixels passed. These three levels of detail are visualized in figure 3.4.

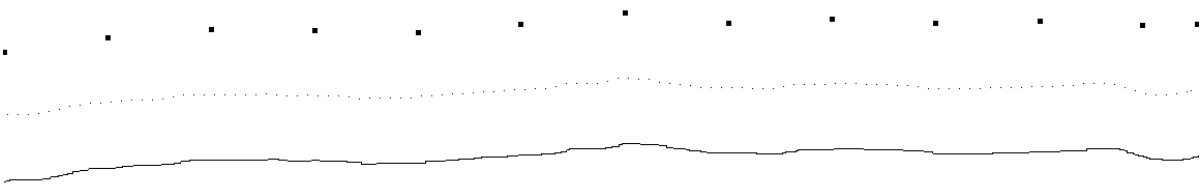

Figure 3.4: Level of detail of a movement trajectory for different cursor movement speeds, created with artificial sampling of pixels that are not removed. The movement is directed from left to right with a constant speed. The top line has 99% of all pixels removed, the second line has 90% of the pixels removed, on the bottom line, no pixels are removed. Please not that in a real measurement, the distance between pixels that are not removed would not be of equal distance, creating a much more irregular pattern.

The image shows that even at high movement speeds, the detector can observe the general movement trajectory. Compared to the most detailed variant however, subtleties in the trajectory are lost and cannot be observed by the detector.

The loss in subtleties is even larger for *cursor speed*. Cursor speed can be approximated by calculating the distance the cursor has moved since the last event and subsequently dividing this number by the time between the last and current event. In figure 3.4, the cursor speed is constant. If the cursor speed increases during the movement, the sampled points have an increasingly large distance between them. This is discussed in more detail in section 4.1. An average speed can only be calculated between two events. Therefore, with high cursor movement speeds only few average speeds can be calculated. Adding to that the uncertainty introduced by the delay between fired events (the delay can range from 5 to 88 milliseconds in 230 samples), the detector finds itself with a limited view on cursor movements. Still, even this limited view is powerful, as will be elaborated on in section 3.4.1.

3.3.3 Clicking

In this category, three events are relevant:

- `mousedown`
- `mouseup`
- `dblclick`

The `mousedown` and `mouseup` are the most interesting click events. The `mousedown` event is fired when the user presses down any standard mouse button. The `mouseup` event is fired when the user releases that mouse button. Both events provide the location of the click.

Taken together, the `mousedown` and `mouseup` events reveal how a user clicks. The time between the `mousedown` and `mouseup` event is the *dwell time* of a click. Together, the `mousedown`, `mouseup` and `mousemove` events reveal whether a visitor moves the mouse while clicking.

The `dblclick` event is an event that only just falls within the category of *action*-based interaction. It fires after two click events have already fired, if the time between the two clicks was smaller than a threshold. This threshold is defined by the operating system when a user clicks, and defined by Selenium when the Selenium webdriver clicks[7]. As the detector can see the individual clicks without the event already, the main contribution of the `dblclick` event is that the detector can observe when the visitor intended to doubleclick. The user aims to click twice within the double click threshold if evoking a `dblclick` event is desired. Knowing the user intended to doubleclick, the detector can analyze the earlier `click` or `mousedown` and `mouseup` events to classify the double clicking interaction properties.

The `dblclick` event can also be used in static fingerprinting. The threshold to regard two clicks as a double click is defined by Selenium to be around 600ms. This is higher than the default of 500ms for Windows[8] and some Linux distributions, making Selenium stand out by firing the `dblclick` event if the delay between clicks is between 500 and 600 milliseconds.

3.3.4 Scrolling

In this category, two events are relevant:

- `scroll`
- `wheel`

[7]The specification states: "The definition of a double click depends on the environment configuration(...)"`https://w3c.github.io/uievents/#event-type-dblclick`

[8]`https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setdoubleclicktime`

The `scroll` event fires when the viewport[9] is moved over the page. In contrast to the `mousemove` event, the `scroll` event is always fired when the viewport moves. This enables the detector to observe the scrolling interaction properties in great detail.

Different methods to scroll in a browser have their own event. When using an arrow key, events for the pressing and releasing of that key are fired. When using the mouse wheel, the `wheel` event is fired. This event allows the detector to observe how far the wheel is turned. In Firefox on Linux, one tick of scrolling with the mouse wheel scrolls 57 pixels.

Firefox offers, and enables by default, a browser feature called "smooth scrolling". When this feature is enabled, the scroll tick is divided into parts by Firefox to let the viewport slide more smoothly over the document. If the feature is disabled, there is just one wheel and one scroll event. The detector can easily detect whether the feature is enabled by the difference in events fired. When it is determined whether the feature is enabled or disabled, scrolling interaction can be analyzed.

3.3.5 Typing

In this category, two events are relevant:

- `keydown`
- `keyup`

The `keydown` and `keyup` events fire when a key is pressed, and when a key is released respectively. The events provide the key for which the event was fired. As such, the detector can observe for every key when it was pressed and released with a precision up to 1 millisecond[10].

When a user types, the typing usually is not limited to one character. *Typing* is therefore considered here as an *action* that involves a continuous input of characters. 'Continuous" should not be interpreted too strictly here, as there may be short pauses in the typing action. Humans sometimes pause while typing as typing and thinking about what to type can not always be done at the same time [ACdSS07]. As pauses are part of the typing process, it is considered part of the same action here.

The detector can observe precisely the manner in which a visitor types by calculating the timings between different key pressed and key releases. The time between a key press and a subsequent release is the *dwell time*. The time between the release of a key and the key press of a (probably different) key the *flight time*. The detector can also count the amount of key presses in a timeframe to calculate the average *typing speed*. Presses of special keys such as the `backspace` and `del` key indicate *typing accuracy*.

Taking these factors together, a detector can get a detailed view of the typing interaction properties of a visitor.

3.3.6 Leaving and entering

In this final category, two events are relevant:

- `visibilitychange`
- `blur`

While evaluating the list of events, two events surfaced that do not fit in either of the four categories of interaction on web pages identified by Krumnow et al. [KJK21]. Those events

[9] The viewport is the part of the document currently visible in the user agent

[10] 1 millisecond is the default value, which can be changed in the Firefox configuration under `privacy.resistFingerprinting.reduceTimerPrecision.microseconds`

are about giving focus to an element on a page, or the page itself, indicating the visitor is multitasking.

The `visibilitychange` event fires when the visibility state of the pages changes for the visitor. The completeness of the visibility state is severely limited. According to the current specification[11], the visibility is only set to hidden if:

"1. The user agent is minimized.
2. The user agent is not minimized, but [the] doc[ument] is on a background tab.
3. The user agent is to unload [the] doc[ument].
4. The Operating System lock screen is shown."

The last point is not implemented in either Firefox or Chrome. This means the detector can only conclude the visitor is not looking at the page if either the visitor minimized the browser window or when a different tab is currently visible to the user.

The `blur` event is similar, but differs slightly. It indicates an HTML element has lost focus. This can happen if another element gets focus, which can happen for example when a user clicks on that element. If the `document`[12] is taken as element, `blur` fires on specific actions. Roughly, it fires whenever the user performs an action outside the web page.

The events allow the detector to see if the visitor performs any actions outside the document, and how often is switched. How these and the previously discussed categories of interaction can be used to classify web bots will be subject of the upcoming section.

3.4 Comparison of simulated and human interactions

In Section 3.3 five categories through which interaction properties can be observed have been discussed. For every category, JavaScript events that provide the detector with information have been identified. In this section, human interaction will be compared to the way of interaction of Selenium for as far as it can be observed through the selected JavaScript events.

3.4.1 Mouse movements

Two key distinctive properties of Selenium mouse movements are the trajectory of the movement and the speed of the movement. Selenium moves in a perfectly straight line between the start and endpoint of the movement. The movement is also nearly instantaneous. This is completely different from human movements, which are curved, shiver and take some time to complete. The difference in trajectory is visualized in figures 3.5 and 3.6

[11] The below list is a verbatim copy of the specification, see `https://www.w3.org/TR/page-visibility/#dom-visibilitystate`

[12] Essentially an element that contains the complete page

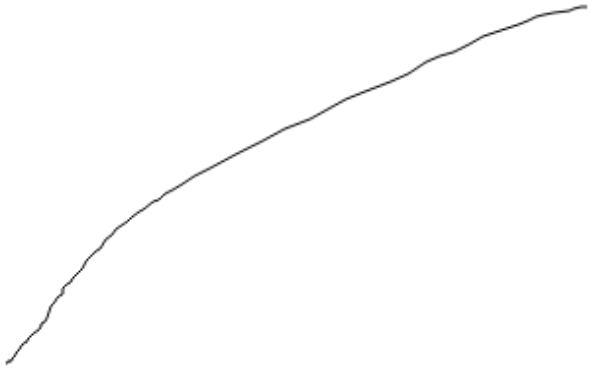

Figure 3.5: A human cursor trajectory.

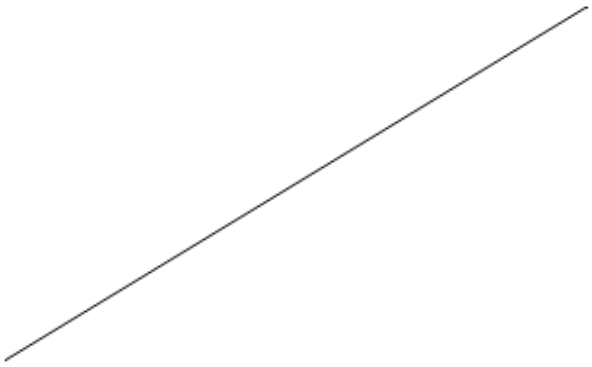

Figure 3.6: A Selenium cursor trajectory.

Two other distinctive properties are the initial location of the cursor and the start time of the actions. For humans, the cursor might be on any location when a new page is loaded. Selenium however, always has the cursor in the top left corner when it starts. Movement and other interactions also only start after Selenium deems the web page ready to receive actions[13]. A detector can exploit this by ensuring the page is displayed to the user, but remains being seen as "loading" by Selenium. A human is likely to start exhibiting interaction if the page is displayed, while Selenium will wait until the page is loaded. Both properties are out of scope, as they are not part of an *action*. Rather, they are properties of combinations of actions, and fall under the *chains of actions* category of interactions.

A final distinctive property is spontaneous cursor movements. Selenium will never move the cursor when it is not instructed to do so. As a result, it will only move as part of an action with a goal, like clicking on a button. Humans may accidentally move the mouse, changing the cursor location without intention. However, for this to be a good web bot detection method, there should not be too many people that do not move the mouse accidentally, as they would all falsely be classified as a web bot. The usability of the method should thus be verified with a number of humans of different demographic populations, which is out of scope for this work.

3.4.2 Clicking

To compare human and Selenium clicking, a simple experiment was performed. In the experiment, 100 clicks are performed by Selenium and a human. The results of Selenium clicks are presented in Figure 3.7 and Table 3.1. The results of the human clicks are presented in Figure 3.8 and Table 3.2.

The distribution of click locations within an element in Figure 3.7 clearly shows the distinctive pattern of Selenium clicks. Selenium always clicks on one out of four pixels in the middle of the element. Figure 3.8, shows the human distribution of clicks. Not a single of the 100 clicks are on one of the four pixels Selenium frequently clicks on. This is to be expected, as with $25 \times 100 = 2500$ pixels in this button, the chance of hitting on of the four pixels in the middle is small, even with the small bias to the middle of the button the human click data shows.

When it comes to clicking, Selenium again stands out by its speed. The dwell time of a click is only 3 milliseconds on average. Compared to the average 93 milliseconds for a human in the experiment, Selenium can not be mistaken for a human when at least a few clicks are provided. Out of 100 human clicks, only one had a dwell time of 3 milliseconds[14]. Selenium on the other hand, had only 2 dwell times larger than 6 milliseconds out of 100 clicks. 96 clicks

[13] `https://www.w3.org/TR/webdriver2/#navigation`, last visited June 3, 2021

[14] This might even have been an error in measuring, as the second lowest human dwell time was 32 milliseconds

had a dwell time of 4 or lower. This property alone can reveal Selenium after 2 clicks with high certainty.

Taken together, these detection features can reveal Selenium with high chance in just one click.

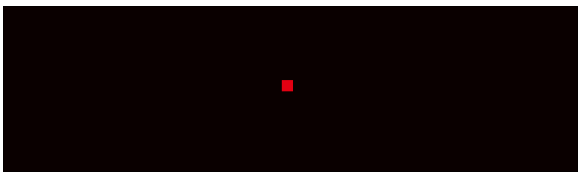

Figure 3.7: Visualization of Selenium click locations within a button. Black indicates the button surface, white indicates pixels on which was clicked and red indicates pixels which were clicked more than once.

	Mean	**SD**
x	49.56 px	0.50 px
y	13.43 px	0.50 px
dwell	2.66 msec	1.96 msec

Table 3.1: Statistics of Selenium clicking interaction.

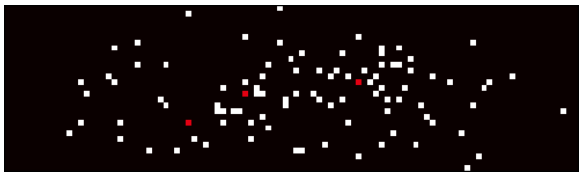

Figure 3.8: Visualization of human click locations within a button. Black indicates the button surface, white indicates pixels on which was clicked and red indicates pixels which were clicked twice.

	Mean	**SD**
x	51.99 px	19.64 px
y	15.21 px	5.85 px
dwell	92.53 msec	18.03 msec

Table 3.2: Statistics of human clicking interaction.

A different method to leverage the `mouseup` and `mousedown` events is to watch for one or more `mousemove` events during the time the mouse button was pressed. Selenium never moves the mouse while clicking, as it is not programmed to do so. Users might move their mouse during a click. To get a reliable test, multiple kinds of mouses would need to be tested, as heavier mouses with low sensitivity settings are less likely to expose this kind of interaction properties. Even more important is the user itself. As larger user studies are out of scope, this is left for future investigation.

3.4.3 Scrolling

In its API, Selenium only directly supports touch based scrolling. As OpenWPM simulates a desktop user, using touch based scrolling is not a preferred way to scroll. It is possible however to use JavaScript to scroll. But when JavaScript is used to scroll, the scrolling movement is instant, can span enormous distances and never needs to be corrected. Additionally, it does not evoke `wheel` events, indicating no mouse or touchpad was used to scroll, making the scrolling method more suspicious.

Although Selenium's way of scrolling is different from a user scrolling with a mouse wheel, it is not easy for a detector to implement detection routines. A human can also evoke scrolling movements that are instant, cover enormous distances and do not need to be corrected via for example the search function in a browser. Using the scroll bar is also possible, and so is using anchor links. The possibilities to use the middle mouse button or arrow keys to scroll further complicate detection. Firefox might also scroll when the page is reloaded. All these methods to scroll can be detected. But in contrast to the other detection methods, all other possibilities

have to be ruled out. This is much more complex than confirming one possibility, as suffices for the other detection methods.

3.4.4 Typing

Like clicking, typing reveals the dwell time of a button press. It also reveals flight time. A short experiment has been performed, in which the Selenium and human `keydown` and `keyup` events are monitored.

The maximum dwell and flight times for Selenium are 3 and 2 milliseconds respectively. This again contrasts to the human dwell time, of which the mean is again 100 times higher. The flight times are not compared, because it is complex to measure flight time for humans, and it is clear without measurements that flight times of Selenium differ from human flight times.

Not only the flight times are different, the order of events also differs between Selenium and fast typing humans. Selenium always evokes a `keyup` event after a `keydown` event. A human typing with two fingers on a low speed will likely show the same pattern, although this should be tested with more than one human. In contrast to Selenium and slow typing humans, a human typing with 10 fingers at a high speed interleaves `keydown` and `keyup` events. Keys are pressed down before earlier pressed buttons are released. In the experiment it is observed that a human typing with 10 fingers on a high speed (600 characters per minute) more often has two buttons pressed at once than no buttons pressed at all. Selenium on the other hand, always releases the previous key before pressing a new one. A human typing with 2 fingers showed, apart from a large difference in speed, almost the same pattern as Selenium in the experiment. A major difference being that for capital letters, two buttons where pressed at the same time by the human, while Selenium never presses more than one key at the same time.

To type a capital letter, it is required on the standard US-international keyboard layout to either press the *Caps Lock*-key, or to hold the `shift`-key while pressing the character that should be capitalized. This reveals yet another property specific to Selenium. Selenium can "press" capital keys, as if they existed on the keyboard. As a result, Selenium can be detected with high accuracy after both a capital and non-capital key have been pressed. As both pressing the *Caps Lock* key and leaving the browser window can be detected, a visitor sending capital and non capital letters without other events is either Selenium or a human using special software, like accessibility tools. Diacritics are also revealing Selenium instantly with high precision. Apart from the problem that Selenium "presses" the key without using modifier keys, an even more severe problem exists. When the `keydown` event is fired, JavaScript includes an attribute with the `code` of the key that was pressed. This attribute is empty when Selenium sends a character with a diacritic like é, while for a human with the US-international keyboard layout the codes are `Quote` followed by `KeyE`.

Although out of scope of this analysis, it is interesting to notice that the keyboard setting of visitors can be inferred via this method, due to the additional key presses required for special keys.

Besides profiling users based on one or two key presses, more advanced profiles of typing can also be made. Alves et al. [ACdSS07] provided statistics on typing properties regarding opening a new sentence or a new word and more special cases in typing a text. These statistics are discussed in more detail in Section 4.4.

Another important aspect to typing is the accuracy of typing. Humans tend to make typing errors and correct them, either by using the backspace key to redo the typing or by moving the cursor and correcting the error. Selenium does not make typing errors, which can be deemed suspicious. Due to time constraints, this is not looked into further in this analysis.

Some types of text input fields elicit special typing rhythms, like username and password fields. As a password is a set of characters that is frequently used, it may be typed much

faster than other texts. It might also be that a password is typed slower, as the user reads the complex password from a post-it. In any case, the typing method will differ from standard text fields in which the user types a text in a natural way. Using and mitigating such advanced detection techniques are left for future work.

3.4.5 Leaving and entering

Events indicate whether the visitor can still see the web page of the detector (`visibilitychange`) and if the web page is still focused on (`blur`). Both events reveal specific interaction of a visitor. Selenium will never fire either of these events, as the task of Selenium is on the page, not outside it. Depending on the web site, human visitors may switch between applications or browser windows. Although the absence of this switching does not definitively proof anything, it may help in creating a risk profile of a visitor.

Some special cases exist though. `visibilitychange` fires if a webdriver (Selenium) controlled window is minimized by the user of the web bot. Selenium will continue its tasks, which can lead to Selenium performing actions on a web page that is known to be invisible to any real user. This trivial to test property will instantly reveal Selenium. It is not likely to happen though, as web bots are implemented to relief a human from having to perform the interaction, so one would expect no user interaction to occur.

A problem with these events might arise in the future, as the draft specification for page visibility more concretely states that a `hidden` state should be returned if the document is not visible to the user: "Return visible if: (...) Any of the doc[ument]'s viewport contents are visible to the user. Otherwise, return hidden."[15]. If a web bot operator runs multiple headful[16] browser instances of OpenWPM, only one is visible to the user. All others are hidden behind the browser instance that is visible, creating a situation where all but one instances can be detected at all times.

[15] `https://w3c.github.io/page-visibility/#dfn-determine-the-visibility-state`

[16] Running the instances headless might solve this problem, but headless instances are more detectable via other properties [Vlo18, JKV19, Goß20]

Chapter 4

Mitigating detection

In Chapter 3 it has been shown that interaction based detection is an accessible and effective method to detect web bots that use Selenium for within-page interaction. In order to reduce the effect of web bot detection on results gathered by OpenWPM, a Python package, *Human-Like Interaction Selenium API* (HLISA), is developed. The aim of the package is to simulation human like interactions, which would place this package between state D2 and D3 of the detector model.

From a technical perspective, the HLISA package replaces the `ActionChains` class[17] of the existing Selenium API for interaction. The set of methods exposed by HLISA is a superset of the existing methods in the `ActionChains` class. An end-user can replace the `ActionsChains` objects with `HL_ActionsChains` objects in existing code to switch from the standard Selenim API to HLISA. This makes OpenWPM harder to detect while the functionality will be equal, except for time duration. The HLISA methods require more time to execute, as humans are substantially slower than bots.

In this chapter, all functions the HLISA replaces are discussed. This is done by listing every interactional property, then the hiding tactic, then implementation level details.

4.1 Mouse movement

4.1.1 Hiding tactic

In section 3.4.1 it was shown the cursor moves of Selenium are easy to distinguish from human cursor moves. To make the cursor movement look more human like, first of all the straight line is replaced by a curved line. The line is based on a quadratic Bézier curve that has a horizontal bias. This means the horizontal movement is too large in the beginning to end up in the right place. This is compensated for at the last third of the curve. This creates a curve that is not just a straight line with a curve, but a curve that changes its direction midway.

Whisenand and Emurian observed that horizontal movements are performed faster [WE97] than vertical movements. Therefore, the bias is made horizontal. This is too simplistic, as many more factors have a role in movement [WE96, WE97, LB13, PT01]. The bias will likely depend multiple factors, and turn into a vertical bias in some occasions.

Additionally, a "shivering" movement is added to the curve. This transforms the curve from a perfect mathematical curve into a more natural curve, with small distortions in all directions. As can be seen in Figure 4.1, a movement with a curve is still clearly different from a human cursor movement as visualized in Figure 4.2.

[17] `https://www.selenium.dev/selenium/docs/api/py/webdriver/selenium.webdriver.common.action_chains.html#module-selenium.webdriver.common.action_chains`

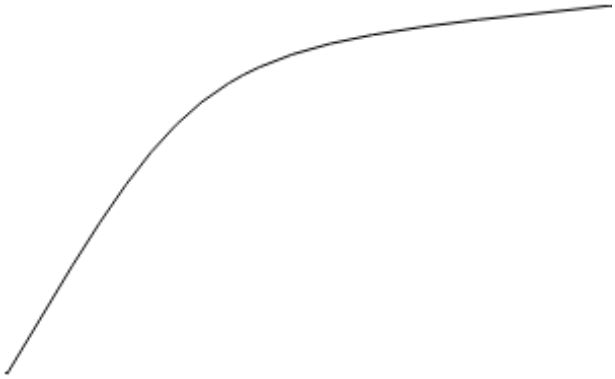

Figure 4.1: A perfect curve.

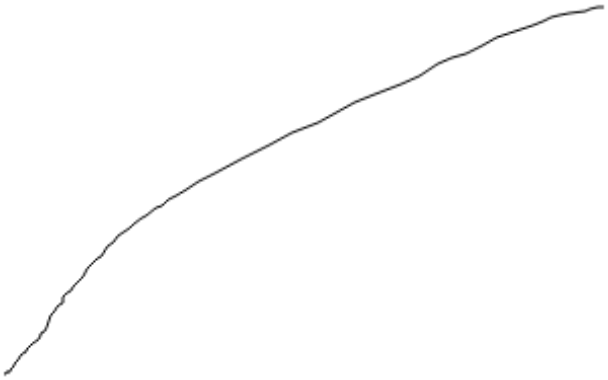

Figure 4.2: A human cursor trajectory.

Finally, the overall movement speed is reduced. Instead of performing a movement in a fraction of a second, it is done at a speed which is easy to reproduce by a human. The movement speed increases until 70% of the curve has been traversed. After that, the movement quickly slows down.

Both the curve direction and movement speed change in the last third of the curve. Together, this simulates a "correction". After an initial large movement in the general direction of the target, the movement is refined and carried out with greater precision (the slower movement) to end up at the target. This correction was observed in the experiments carried out for Chapter 3 and has been covered by Grahma et al. [GM96].

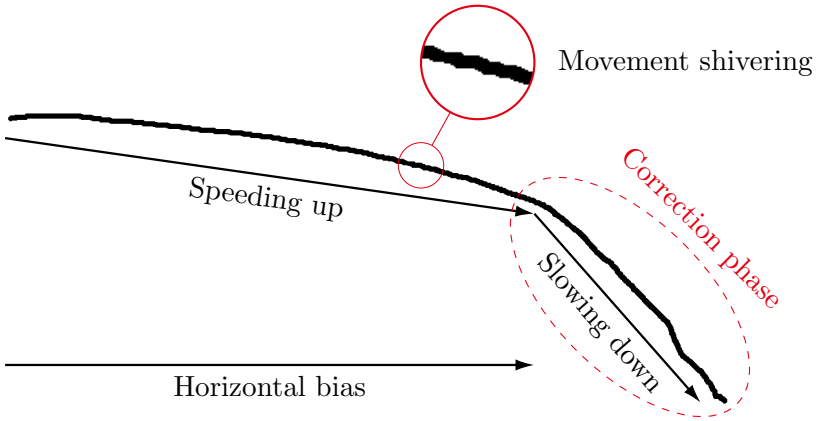

Figure 4.3: Overview of characteristics of mouse movements simulated by HLISA.

There are plenty of other factors in mouse movement, like overshoot [PT01], angle of approach [PT01, WE97, WE96, LB13], target shape [WE97], target size [PT01] and direction of movement, either in the difference between horizontal and vertical [LB13] or between left and right directed movements [PT01]. Individually they might already provide a detector with enough information to detect humans with a reasonable certainty. Combined they can provide the detector a rich set of aspects to detect humans by their specific characteristics. Certainly, the mouse movement hiding tactic provided here is not sufficient to counter all detectors in state level D2. Web bots that use these tactics by mouse movements are harder to detect nonetheless.

4.1.2 Implementation

The curve in the movement is based on a quadratic Bézier curve. The first and last point are given, but the intermediate point (the only control point of this type of Bézier curve) needs to be calculated. To create the described horizontal bias, the intermediate point is hardcoded to be at 50% of the height and 82% of the width of the total distance. The 82% was selected as it created a movement visually similar to the observed human curves. As mentioned in the previous section, this can be improved upon. For reference: to create a straight line, the intermediate point should be at 50% height and 50% width. For a curve without bias, the height should be equal to the width. The curve will not go over the intermediate point, it will only go into that direction as if it is attracted by the point. Only in the 50% height and 50% width case, the line will go over the point. In this case the point has no effect on the curve.

The shivering movement is created by sampling from a random distribution. This distribution, with mean 0 and standard deviation 0.6 mostly returns values between -2 and 2. For every part of the movement, this random value is added to the curve creating a movement deviating away, but always coming back to, the perfect mathematical curve. The random values that are added to the curve are recorded and in the last part of the movement they are compensated for. Because of this compensation, the movement ends at the location that was given to the movement function. The compensation becomes stronger as the movement progresses, which corresponds to human movement which also is more precise near the end. A movement trajectory produced by HLISA is presented in figure 4.4.

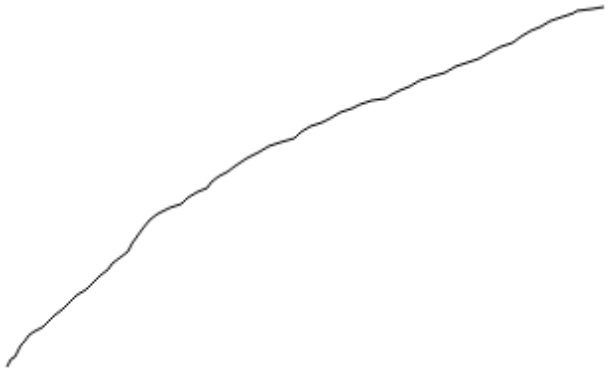

Figure 4.4: Cursor trajectory by HLISA.

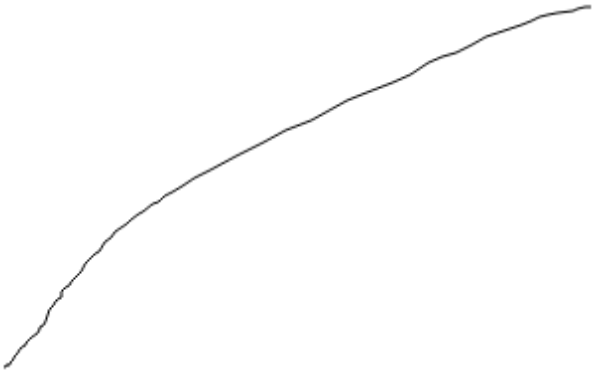

Figure 4.5: A human cursor trajectory.

The mouse movement speed is based on two factors. The first factor is the location of the intermediate point. The intermediate point is passed when half of the movement time is over. Therefore, if the location of the intermediate point is further than 50% of the total horizontal distance, more horizontal movement will be covered in the first half of the movement as in the second. This is an unwanted side effect of the Bézier curve logic, as it makes that the movement speed is not completely dependent on the other factor which is specifically created to control the mouse movement speed. Although this is a disadvantage, the movement speed can still be controlled by adjusting the second factor in order to account for the movement speed difference created by the intermediate point. As the intermediate point is fixed, the unwanted influence is also fixed and can therefore be accounted for without complicated measures.

The second, and main factor that controls the mouse movement speed is the definition of the sampling function. This is the main way to control the movement speed. n points that are on the Bézier curve are calculated. The distance between every point that is calculated is equal, except for the difference that is caused by the first factor discussed in the previous paragraph. m points are then sampled from the n points by the sampling function. The sampling function controls the speed, because $speed = \frac{distance}{time}$, and $time$ is fixed, while $distance$ is controlled by

the sampling function as it determines how much distance exists between points. Not selecting a point increases the distance between the previously selected point and the next point that will be selected.

The sampling function can be replaced by any mathematical function. To get a constant and low speed, all points are selected. To get a constant and high speed, every 4^{th} point is selected. Acceleration and deceleration are introduced by selecting increasingly many points for a decelerating movement, and increasingly less points for an accelerating movement. In the current implementation, after every a points, a new point is selected. a is increased with 5% after a new point is selected. This is done until 70% of the points are processed, after which a is decreased with 10% after each new new point that is selected. This creates a movement that first accelerates in an exponential fashion, until it quickly decelerates in the last part of the movement, also in an exponential fashion, as described in the hiding tactic section.

4.2 Clicking a button

4.2.1 Hiding tactic

In Section 3.4.2 it has been shown Selenium `click` actions can be easily detected. To prevent detection, the Selenium `click()` command is replaced with a more realistic version in HLISA. If an element is passed to the `click` command, the new command first moves the mouse to a random position on the given element. Thereafter it clicks and holds the button for a random time in a given interval.

For the location and press duration, first a random distribution was used. As can be seen in Figure 4.6 this is easy to recognize as simulated interaction. One can simply compare the amount of clicks in the border region with the amount of clicks in the center of the button, and the new bot detection method is operational. As state level D3 is the target level of the HLISA, a different approach was taken. The random numbers are taken from a normal distribution.

The distribution was acquired by recording human mouse clicks[18] on a web page specifically set up for this purpose. The unrealistic environment may contribute to the distribution being not representative. But a normal distribution for mouse clicks is not completely realistic to begin with as Barik et al. [BHRJ12] have shown that human clicks are not always normally distributed with the center of the button as mean. The research by Barik et al. also showed that standard deviations depend on specific buttons and other factors, which cannot all be accounted for given time constraints.

Besides the distribution, this way of clicking has a more important limitation. Humans sometimes miss the button and click besides it. This is not accounted for in this HLISA command. It could be extended to do so, but this is difficult as the program needs to guarantee the misclick does not cause any side effects.

4.2.2 Implementation

To get a location in an element to click on, a function is defined that receives an element and returns coordinates within that element. When querying the bordering coordinates of an element, the Selenium API and JavaScript return a square box around the element. But elements can have different shapes. For example, a button can be round. Tested with `style="border-radius: 12px;"`, 8852 of 10,000 randomly generated coordinates were in the button and 1148 outside the button when using the borders as given by JavaScript.

To account for this, it is verified whether the chosen point is in the button. If the point is not in the button, a new point is sampled and the check is performed again. This happens up to 10 times, after which it is assumed that the button is not clickable at all. The function can

[18] From one user only

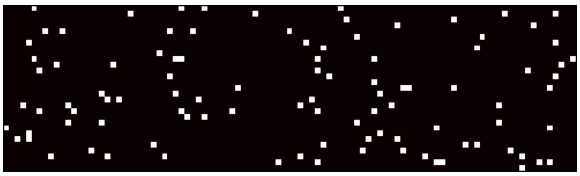

Figure 4.6: Random distribution visualized. Black indicates the button surface, white indicates pixels on which was clicked and red indicates pixels which were clicked twice.

	Mean	SD
x	50.99 pixels	30.37 pixels
y	14.81 pixels	8.40 pixels
dwell	112.11 msec	28.64 msec

Table 4.1: Statistics clicking interaction based on a random distribution.

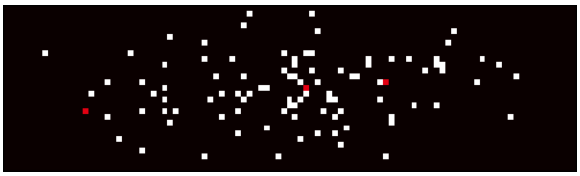

Figure 4.7: Normal distribution visualized. Black indicates the button surface, white indicates pixels on which was clicked and red indicates pixels which were clicked twice.

	Mean	SD
x	50.45 pixels	18.55 pixels
y	14.13 pixels	5.45 pixels
dwell	103.36 msec	16.36 msec

Table 4.2: Statistics of clicking interaction based on a normal distribution.

thus be expected to fail approximately once in a billion clicks. The functionality also prevents the web bot from clicking on an element that is not visible. This sometimes happens if the element is hidden or overlapped by another element, for example when an image is placed over a button, or when a buttons visibility is set to hidden. In those instances, Selenium clicks on the button while HLISA does not click but issues a warning to the web bot operator instead.

4.3 Scrolling

4.3.1 Hiding tactic

Selenium does not provide commands to scroll. Selenium users can use JavaScript to scroll. To make using HLISA easy, additional commands to scroll are implemented in HLISA. These follow the `scrollTo()` and `scrollBy()` JavaScript functions in syntax to make migration easy. The commands can not be used like normal `ActionChain` commands as is usual in Selenium, as the commands are converted to JavaScript. In contrast to `ActionChain` commands, JavaScript calls are executed directly, which makes them incompatible.

The JavaScript `scrollTo()` and `scrollBy()` functions take either one `options` argument or an `x` and `y` argument. The `options` argument allows to specify the scrolling method. Only the standard scroll method is implemented, leaving the argument with only one option. For this reason, the argument is left out completely.

To be consistent with the JavaScript functions, the new `scrollTo()` and `scrollBy()` functions should take two arguments, `x` and `y`. The `x` argument is to specify horizontal movement, while the `y` argument is to specify vertical movement. For now, the new `scrollTo()` and `scrollBy()` commands accept both the `x` and `y` arguments, but moving horizontal (using the `x` argument) is not yet supported. It depends on how the site is built whether the user is able to scroll horizontal with the mouse, or the scrollbar has to be used, resulting in different interaction properties. As scrolling horizontal is rare, this functionality is left unsupported for now.

4.3.2 Implementation

To simulate human like mouse scrolling interaction, some factors have to be taken into account:

1. One mouse scroll tick scrolls a predefined amount of pixels, 57 in Firefox,
2. There is some time between every scroll tick, depending on the scrolling speed of the user,
3. After a couple of scroll ticks, the user has to replace the finger used to scroll back to the top or bottom of the scroll wheel in order to continue scrolling.

The newly implemented HLISA `scroll_to()` and `scroll_by()` commands both perform the actual scrolling by calling a newly defined scroll function. This function is not accessible in the public part of HLISA. The function takes the amount to be scrolled and starts scrolling in even steps of 57 pixels with a pause of (0.05 + `r` / 200) milliseconds in between the steps, where `r` is a random value ranging from 0 to 1. Although not based on studies or observational data, the formula with these constants creates interaction that seems to be quite human like. After seven scroll ticks, a random pause with a mean of 500 milliseconds and a standard deviation of 100 milliseconds is added to create the finger replace effect.

A limitation of this implementation is that in standard Firefox, smooth-scrolling is enabled. If this is not disabled in the settings of Firefox, the scrolling interaction looks different. It is easy to detect whether the smooth-scrolling is enabled, and as smooth-scrolling is default, it can be suspicious to use a different scrolling method.

Also, because one scroll tick scrolls a fixed amount of pixels, it is not possible to scroll less than this fixed amount without looking suspicious. Therefore, the amount of pixels that is scrolled can differentiate from what was given as parameter up to the fixed amount minus one.

As was stated in Section 3.4.3, detecting non-human like scrolling is difficult. But if the HLISA is to be improved, implementing smooth-scrolling is one of the most important features to implement first.

4.4 Typing

4.4.1 Hiding tactic

To counter the recognition of Selenium by the dwell time, a delay is added between the pressing and releasing of a key. This is the same solution as is used for clicking. Additionally, as typing is seen as one *action*, flight time needs to be accounted for as well. This is done by adding a delay between all key presses and subsequent key releases. On certain moments, extra delay is added, which will be elaborated on in Section 4.4.2.

The other mayor problem with Selenium typing interaction is missing modifier key presses to create certain characters. The HLISA package prevents detection by pressing `shift` keys during the typing of special characters that for the US-International keyboard layout require the `shift` key to be pressed. The following characters that require a pressed `shift` key are supported:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ! @ # $ % ^ & * ( ) _
+ { } | : > ?
```

This means many characters are not supported, as for example all characters with diacritics require the pressing of modifier keys in the US-International keyboard layout. For characters with diacritics there is the additional problem of the empty `code` attribute. This is always instantly detectable and can not be circumvented by pressing additional keys. As such, those keys are not supported by HLISA.

Location		Time (seconds)		Implemented
Natural language	**Formal**	**Mean**	**Std dev**	
Opening a sentence	.␣^a	1.3	1.0	Yes
Closing a sentence	a^.	1.7	0.7	Yes
After closing a sentence	.^␣a	0.6	0.4	Yes
Before comma	a^,	1.8	1.0	No
After comma	,^␣a	0.17	0.04	No
New word after comma	,␣^a	0.6	0.36	No
Opening a word	a␣^a	0.47	0.21	Yes
Within a word	a^a	0.21	0.03	Yes
Closing a word	a^␣a	0.20	0.08	Yes

Table 4.3: This is an excerpt of Table 3 of the article of Alves et al. [ACdSS07]. The syntax is, according to the description of the original table: "'a' stands for any letter, "␣" for spacebar, and "^" for absolute pause. Periods and commas are indicated as such." The most right column shows whether the timing is used in the HLISA implementation.

4.4.2 Implementation

The dwell and flight times are random numbers from a normal distribution. The distribution is again based on a small experiment. The additional delays that are added on special moments are based on research by Alves et al. [ACdSS07]. They included a table with special moments and corresponding statistics. In Table 4.3, which is an excerpt of a table from the study of Alves et al. it is shown which of their findings have been implemented in HLISA.

There are more factors that introduce delays described in the work of Alves et al., like pauses after a certain number of words, but they have not been implemented as they depend on the cognitive load of the writer. Delays before and after commas have not been implemented although it can be considered within scope. These pauses were left out because the typing speed is already fairly low, even though the statistics of the group "Fast" have been used.

All delays are based on random numbers from a normal distribution. Because such numbers can be negative, the number selection process is slightly altered. The random number is used, unless it is smaller than a minimum. If it is smaller, a random small number is added to make the original random number larger than the minimum. This is done in such a way that the original random number can not become larger than the mean. The values that are added are uniform random numbers to make sure the shape of the distribution is not altered too much.

4.5 Leaving and entering

Although the events that indicate whether a user has hidden the browser tab or has selected another screen are interesting, they do not pose a high threat to simulators. OpenWPM with Selenium never switches away from the document it is interacting with. This can be used in a risk assessment by the detector, but not switching away does not prove a visitor is a bot.

To prevent web bot detection by fingerprinting, it is advisable for the simulator to use headful OpenWPM [JKV19]. But if this window is minimized by the user, and Selenium is still performing actions, a web bot detection script can immediately detect Selenium is active. Therefore, a user should never minimize such a window. Bringing another window in the foreground has no effect, as the `blur` event is not fired in this case when Selenium is active.

When the new page visibility draft[19] comes into effect, behavior of events may change. Depending on the change, countermeasures might have to be taken.

[19] `https://w3c.github.io/page-visibility/#dfn-determine-the-visibility-state`

Chapter 5

Validation

5.1 First validation

In Section 3.4, five categories of interaction that can be used to detect Selenium have been discussed. In Chapter 4 countermeasures have been presented to mitigate detection. To test the performance of these countermeasures, a simple detector was implemented that detects Selenium by the properties discussed in Section 3.4. As it detects interaction properties specific to Selenium, the detector has level D1 according to the detector model. Both Selenium and HLISA are tested against this detector to see if the assumptions hold.

5.1.1 Selenium detector implementation

The simple detector, called "Selenium detector" is an extension of the DetectOpenWPM project[20] by Krumnow et al. [KJK21]. It implements tests for three out of five categories of interaction: clicking, moving the mouse and typing. Scrolling was not implemented, as this is considerable more difficult than the other four. As was stated in Section 3.4.3, observing a Selenium-like scroll is not enough. All other possible causes of the scrolling event have to be ruled out. Leaving and entering is also not tested for, as the suspicious behavior is only triggered in combination with a mistake of the bot operator, which cannot be tested for in this small experiment. All implemented tests will now be discussed per interaction category.

Test for Selenium mouse movements There is quite a lot that can be tested in mouse movements. One property of movement interaction should be enough to detect Selenium. Therefore, it is tested whether the mouse movement consist of one straight line. On a web page, two HTML buttons are presented. The test consist of clicking on both buttons. The line between the buttons is monitored. Between every two `mousemove` events, the slope of that movement is calculated. If the slope differs 0.03 or more from the last slope, it is considered to be a non-straight line. In that case, the Selenium detector gives the verdict "OpenWPM detected".

Test for Selenium clicking Mouse clicks that are either less than 2 pixels out of the center of a button, or which have a dwell time lower than 5 milliseconds are marked as "suspicious". When the amount of suspicious clicks is higher than 20%, the verdict "OpenWPM detected" is given, otherwise the verdict "OpenWPM not detected" is given. At least 6 clicks have to be performed before a verdict is given, because before 6 clicks one click will immediately cause the percentage to be 20% or higher.

[20] Which will be published after responsible disclosure `https://github.com/bkrumnow/DetectOpenWPM`

Test for Selenium typing For typing, the dwell and flight times are monitored. If the dwell time is lower than 2 milliseconds, or if the flight time is lower than 5 milliseconds, a button press is marked as suspicious. At least 3 presses have to be performed before a verdict is given. Again, OpenWPM is said to be detected if the amount of suspicious presses is larger than 20%.

Limitations

The tests trigger quickly, as a low number of suspicious moves already leads to a detection verdict. But the findings from the previous chapters have been followed, and as such false positives are statistically unlikely to occur.

Unfortunately, in special circumstances false positives may still occur. Special hardware that emulates standard interactions might evoke such false positives. Most prominently, accessibility tools either in software or hardware might trigger detection. Think of a speech recognition tool that types text into a text box. Or a virtual keyboard. Onboard[21], a virtual keyboard "useful for tablet PC users and for mobility impaired users", was tested in the typing test and got the verdict "OpenWPM detected" 5 out of 10 times. This shows that this has to be accounted for in detectors that are used in production, to keep the web accessible.

5.1.2 Results

To find the false positive and false negative rates of the detector, two separate experiments were performed. They are discussed below. Two implementation problems surfaced during the experiments. The mouse movement test and the test that evaluates the location of a click in the button did not work due to a programming mistake in combining the tests. These two tests have been left out.

False positives testing

First it is tested whether the detector falsely classifies a human as a web bot. The visitor is classified as a web bot by the detector if one out of the three tests[22] returns the verdict "OpenWPM detected". For a human with either Firefox and Chromium, this happens 0 out of 10 times. The results are shown in Table 5.1.

Verdict	**OpenWPM interaction API**		**Browser used by human**	
	Selenium	**HLISA**	**Firefox**	**Chromium**
Detected	10	0	0	0
Not detected	0	10	10	10

Table 5.1: Closed test results for false positives.

False negative testing

To test for false negatives, the process was repeated but the classification method was changed. The visitor is classified as a web bot only if all of the three[22] tests return the verdict "OpenWPM detected". This change will make the detector more prone to give a false negative. But no false negatives were given, as can be seen in Table 5.2.

[21] `https://launchpad.net/onboard`

[22] Two tests in practice, as the mouse movement test was disabled.

Verdict	OpenWPM interaction API		Browser used by human	
	Selenium	HLISA	Firefox	Chromium
Detected	10	0	0	0
Not detected	0	10	10	10

Table 5.2: Closed test results for false positives.

5.2 Google reCaptcha

In order to perform a small but real-world test, Google reCaptcha[23] v2 with a checkbox, called simply reCaptcha in the remainder of this chapter, was tested for its interaction based detection capabilities. In contrast to a classical CAPTCHA test, reCaptcha offers the visitor a checkbox that can be clicked. Based on "advanced risk analysis techniques"[23], a user might pass the CAPTCHA just by clicking the checkbox.

reCaptcha is widely used and has been subject of research in the past, among others by Sivakorn et al. [SPK16]. They concluded Google's tracking cookies play a crucial role in the reCaptcha service. After creating a system that automatically creates cookies that seem to originate from a legitimate user, Sivakorn et al. concluded that regardless of IP or size of browsing history, the first reCaptcha that can be passed with just clicking in the checkbox appears after 9 days.

This method was repeated manually. A new virtual machine was used. Browsing was done manually in a manner similar to the reported method of Sivakorn et al. After 14 days, the first reCaptcha was passed by only clicking in the checkbox. As browsing was not done every day, it might be that the threshold for reCaptcha to trust a cookie still is 9 days, as is reported in the paper of Sivakorn et al. After those 14 days, 99 cookies clearly belonging to Google services are present on the virtual machine. Even though sometimes a reCaptcha can be passed just by clicking the checkbox, in most cases the reCaptcha puzzle has to be solved.

With the cookie in place, the role of interaction can be evaluated.

reCaptcha and simulated interaction. Sivakorn et al. used different mouse interaction methods, going as far as using JavaScript functions to click the reCaptcha checkbox. They reported the used interaction method does not have a negative influence on the risk analysis.

This was in 2016 and before Sivakorn et al. disclosed their findings to the reCaptcha team. The reCaptcha team is reported to have altered reCaptcha in response to the disclosures [SPK16]. On the introduction of reCaptcha, the reCaptcha product manager said "even the tiny movements a user's mouse makes as it hovers and approaches a checkbox can help reveal an automated bot"[24]. Did this interactional detection method find its way to the new reCaptcha version?

To answer this question, standard Selenium was used in combination with OpenWPM 0.15.0, `stealth-extension` by Goßen [Goß20] and the created Firefox profile (which contains the Google cookies). OpenWPM was instructed to visit `https://my.malwarebytes.com/en/login/`, fill in fake credentials and finally to click on the reCaptcha checkbox. This checkmark appeared and no puzzle had to be solved 3 out of 3 times. For completeness, the test was repeated with the standard Selenium API replaced by HLISA, which had the same result.

The tests were also performed without using `stealth-extension` by Goßen [Goß20]. This often resulted in some sort of super-reCaptcha, a standard reCaptcha but instead of one or two puzzles, 15 puzzles have to be solved before one is allowed to pass the reCaptcha. Later,

[23] `https://developers.google.com/recaptcha/`

[24] `https://web.archive.org/web/20151002161845/http://www.wired.com/2014/12/google-one-click-recaptcha/`

when the cookies in the profile were 4.5 months old, the reCaptcha on the test site could also be passed without the `stealth-extension`.

It was also possible to pass the reCaptcha from another computer, in a private window, (hence, no cookies, but the same IP-address) in standard Firefox, with human interaction. After changing the IP-address from residential to a campus address, passing the reCaptcha was not possible without solving a puzzle.

In a final test, the screen resolution and IP-address were changed[25]. The standard Selenium API was used, and the `stealth-extension` was not installed. Still, reCaptcha let the bot go through just by clicking on the checkbox. Apparently, profiling by cookies is still one of the most important factors in reCaptcha.

[25] Again, from residential to campus

Chapter 6

Conclusion

Although other aspects of web bot detection prevention have received considerable attention, the threat of web bot detection by considering interaction properties, as well as the mitigation of this threat, have not.

In Chapter 3 it has been shown that standard JavaScript events provide a rich source of information about visitor interaction properties to web site owners. Furthermore, interaction properties for standard Selenium based web bots are clearly distinct from standard human interaction properties. Simple tests can reveal a web bot that uses the Selenium interaction API, defeating other measures like modifying the fingerprint or other distinct features like the `webdriver` attribute. This reveals a large set of possibilities for web bot detectors – and the dangers for web bot operators who need to remain undetected.

In Chapter 4, HLISA, a replacement for the standard Selenium interaction API, was presented. Although HLISA can be improved upon, the difference between the standard Selenium interaction API and HLISA is significant.

This is supported by findings in Chapter 5, in which a simple detector detects the standard Selenium interaction API with high accuracy, while HLISA remains undetected. With the same detector it is also shown that when implementing a web bot detector, care has to be taken to not falsely identify humans using special software or hardware tools as web bots, to keep the web accessible.

In an experiment concerning Google's reCaptcha, interaction properties did not seem to be evaluated. This is in line with earlier findings. It is interesting that such a widely used security tool by a resource-rich entity does not include a simple yet effective source of information in its consideration. To what extend other services use interaction properties has not been evaluated.

Future work. Although earlier work indicates interaction properties are used to detect web bots, it is unclear whether this detection method is widely used. It is hard to determine this, as web bot detectors keep their methods secret and many other factors influence detection. If interaction properties are shown to be used, the effectiveness of HLISA can be evaluated on the identified web bot detectors. Apart from how effective HLISA is in practice, it is clear that HLISA can be improved in multiple ways. This has been discussed in earlier sections in more detail, and is summarized in Table C.1 in Appendix C.

Bibliography

[ACdSS07] Rui Alves, São Castro, Liliana de Sousa, and Sven Strömqvist. Influence of typing skill on pause-execution cycles in written composition. *Writing and Cognition: Research and Applications*, 01 2007.

[ASLN20] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. Web runner 2049: Evaluating third-party anti-bot services. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24-26, 2020, Proceedings*, volume 12223 of *Lecture Notes in Computer Science*, pages 135–159. Springer, 2020.

[AT07] Ahmed Awad E. Ahmed and Issa Traoré. A new biometric technology based on mouse dynamics. *IEEE Trans. Dependable Secur. Comput.*, 4(3):165–179, 2007.

[BHRJ12] Titus Barik, Brent E. Harrison, David L. Roberts, and Xuxian Jiang. Spatial game signatures for bot detection in social games. In Mark Riedl and Gita Sukthankar, editors, *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-12, Stanford, California, USA, October 8-12, 2012*. The AAAI Press, 2012.

[Bor16] Frederik Zuiderveen Borgesius. Singling out people without knowing their names - behavioural targeting, pseudonymous data, and the new data protection regulation. *Comput. Law Secur. Rev.*, 32(2):256–271, 2016.

[CGK+13] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. Blog or block: Detecting blog bots through behavioral biometrics. *Comput. Networks*, 57(3):634–646, 2013.

[EN16] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1388–1401. ACM, 2016.

[Fit54] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology*, 47(6):381, 1954.

[GM96] Evan D. Graham and Christine L. MacKenzie. Physical versus virtual pointing. In Bonnie A. Nardi, Gerrit C. van der Veer, and Michael J. Tauber, editors, *Conference on Human Factors in Computing Systems: Common Ground, CHI '96, Vancouver, BC, Canada, April 13-18, 1996, Proceedings*, pages 292–299. ACM, 1996.

[Goß20] Daniel Goßen. Design and implementation of a stealthy OpenWPM web scraper. `http://www.open.ou.nl/hjo/supervision/2020-d.gossen-bsc-thesis.pdf`, 2020.

[GWXW09] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, and Haining Wang. Battle of botcraft: fighting bots in online games with human observational proofs. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 256–268. ACM, 2009.

[HCBM02] Ken Hinckley, Edward Cutrell, Steve Bathiche, and Tim Muss. Quantitative analysis of scrolling techniques. In Dennis R. Wixon, editor, *Proceedings of the CHI 2002 Conference on Human Factors in Computing Systems: Changing our World, Changing ourselves, Minneapolis, Minnesota, USA, April 20-25, 2002*, pages 65–72. ACM, 2002.

[ITK+16] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 743–758. IEEE Computer Society, 2016.

[Jan21] Mitchel Jansen. Recognising client-side behavioral detection of web bots. 2021.

[JKV19] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. Fingerprint surface-based detection of web bot detectors. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 586–605. Springer, 2019.

[KC18] Peter Krátky and Daniela Chudá. Recognition of web users with the aid of biometric user model. *J. Intell. Inf. Syst.*, 51(3):621–646, 2018.

[KJK21] Benjamin Krumnow, Hugo Jonker, and Stefan Karsch. All browsers are equal... or is openwpm less equal than others? 2021.

[LB13] Byungjoo Lee and Hyunwoo Bang. A kinematic analysis of directional effects on mouse control. *Ergonomics*, 56, 09 2013.

[Noo19] Robbert Noordzij. Synthetic fragmentation experiments using wildfragsim. 2019.

[PA97] Réjean Plamondon and Adel M. Alimi. Speed/accuracy trade-offs in target-directed movements. *Behavioral and Brain Sciences*, 20(2):279–303, 1997.

[PPLC06] KyoungSoo Park, Vivek S. Pai, Kang-Won Lee, and Seraphin B. Calo. Securing web service by automatic robot detection. In Atul Adya and Erich M. Nahum, editors, *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, pages 255–260. USENIX, 2006.

[PT01] James Phillips and Thomas Triggs. Characteristics of cursor trajectories controlled by the computer mouse. *Ergonomics*, 44:527–36, 05 2001.

[RD15] Nathan Rude and Derek Doran. Request type prediction for web robot and internet of things traffic. In Tao Li, Lukasz A. Kurgan, Vasile Palade, Randy Goebel,

Andreas Holzinger, Karin Verspoor, and M. Arif Wani, editors, *14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015, Miami, FL, USA, December 9-11, 2015*, pages 995–1000. IEEE, 2015.

[SM04] R. William Soukoreff and I. Scott MacKenzie. Towards a standard for pointing device evaluation, perspectives on 27 years of fitts' law research in HCI. *Int. J. Hum. Comput. Stud.*, 61(6):751–789, 2004.

[SPK16] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. I am robot: (deep) learning to break semantic image captchas. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 388–403. IEEE, 2016.

[SRMN17] Grant Storey, Dillon Reisman, Jonathan R. Mayer, and Arvind Narayanan. The future of ad blocking: An analytical framework and new techniques. *CoRR*, abs/1705.08568, 2017.

[Vlo18] Gabry Vlot. Automated data extraction: what you see might not be what you get, 2018.

[WE96] Thomas G. Whisenand and Henry H. Emurian. Effects of angle of approach on cursor movement with a mouse: Consideration of fitt's law. *Computers in Human Behavior*, 12(3):481–495, 1996.

[WE97] Thomas G. Whisenand and Henry H. Emurian. Analysis of cursor movements with a mouse. In Gavriel Salvendy, Michael J. Smith, and Richard J. Koubek, editors, *Design of Computing Systems: Cognitive Considerations, Proceedings of the Seventh International Conference on Human-Computer Interaction, (HCI International '97), San Francisco, California, USA, August 24-29, 1997, Volume 1*, pages 533–536. Elsevier, 1997.

[WLS19] Shengye Wan, Yue Li, and Kun Sun. Pathmarker: protecting web contents against inside crawlers. *Cybersecur.*, 2(1):9, 2019.

[Woo99] Robert Sessions Woodworth. *The Accuracy of Voluntary Movement*. The Macmillan Company, 1899.

[YVD+19] Kai-Cheng Yang, Onur Varol, Clayton A. Davis, Emilio Ferrara, Alessandro Flammini, and Filippo Menczer. Arming the public with AI to counter social bots. *CoRR*, abs/1901.00912, 2019.

Appendices

Appendix A

Ethical considerations

Usage of HLISA for malicious purposes. Multiple parties can profit from HLISA. The intended party, researchers using OpenWPM to conduct studies on privacy and security on the internet, would profit from being able to process sites as they are presented to human users. The results of their studies will be influenced less by sites that serve deviating content to bots.

Web bot operators with malicious intent can also profit from HLISA. For readability, they are split into two groups and discussed separately.

The first group consists of high threat level parties: parties with large budgets and in-house expertise. Their goals, such as manipulating the public opinion [YVD+19] requires bots to go undetected. State actors, which have the budget and expertise, might be interested, and have been accused in the media, to perform such activities [ITK+16] The consequences of such activities can be considerable. But these parties do not gain much by the publication of HLISA, as they can easily implement such a library themselves.

The second group of malicious parties have a low to moderate budget and a similar level of expertise. They are individuals or small groups who want to bypass bot detection with a specific goal such as scraping a website. These actors might not have a lot of time, budget or they might lack the specific knowledge, as web scraping is only a side interest to them. This group may be quite large if we assume at least part of the 133,000 views on a question on how Google reCAPTCHA v2 works[26] are from actors trying to circumvent this bot detection measure.

According to an inventory made in Appendix C, if HLISA was to be published it would be among the most easy to use libraries, and the most complete library in regard to functionality. As such, malicious parties of the second group can profit from its publication. But in practice, it is unclear in which circumstances a party will profit from HLISA. The other simulators evaluated in Appendix C are often made to bypass Google's reCaptcha, but in Chapter 5 it has been shown that reCaptcha still does not utilize interaction properties.

The notion that malicious parties might profit from the publication of HLISA, although it is unclear in which regard, should not be taken lightly. Therefore, HLISA will only be made public when the subject of hiding interaction properties becomes more widely known to researchers performing large-scale measurements on the web, for example after a publication about the subject.

Accessibility of the web. In this work, detection of web bots by means of interaction properties has been discussed several times. Although it is trivial to detect web bots using this methods, detectors should be cautious to not falsely classify people using accessibility tools as web bots. The methods described to detect web bots are not ready for use in production environments, and should not be used in those environments.

[26] `https://stackoverflow.com/questions/39422453/human-like-mouse-movements-via-selenium`

Unintended use of the model. HLISAis made to be difficult to distinguish from humans. It is therefore made to appear like the most standard web browsing human. As a result, it not representative of a real human. Furthermore, the statistics on human interaction presented in Chapter 3 are not the result of an experiment with many people from diverse groups, and are inadequate to be used as such.

Appendix B

A selection of browsing events

- **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

Appendix C

Comparison of simulators

Before creating HLISA, an inventory of publicly available code and libraries with similar functionality was made. The inventory is limited to code and libraries that can be easily found on the public internet.

Ease of use. Only two libraries besides HLISA can be imported and used with Selenium without needing to modify anything. But these libraries are limited in their functionality. The other pieces of code and libraries would need to be rewritten to work with Selenium as they are primarily a solution for a different problem, or because it is a code example and not a full solution. In most cases, the code can best be used as inspiration in a completely new codebase, reusing only the parts that are most difficult to implement from these libraries. As the libraries or code pieces are often of high quality, this would greatly benefit the writer of a new library; but a lot of work will still need to be done. As has been discussed in Chapter 4, managing Selenium specific problems is also a time consuming part of implementing a library to mitigate bot detection.

Completeness. None of the evaluated libraries are complete. At most, two categories of interaction are supported per library. Within a category of interaction, the libraries often lack simulation of features that are easy to use for detectors. HLISA supports all four identified categories of interaction, but also lacks certain features within the categories. Most notably, in contrast to HLISA, BezMouse supports the simulation of accidental right, double or no mouse clicks. But these functions can be problematic when programming the web bot, as the program no longer is deterministic. For the intended purpose of BezMouse, like aim bots, this is not a problem. But when crawling it can be a problem.

In conclusion, HLISA is similarly easy to use as the ready to use publicly available libraries for simulating human interaction. But in contrast to these and other publicly available libraries, it is most complete. This holds for the supported categories of interaction as well as functionality to simulate interaction realistically. Finally, in contrast to the other libraries, HLISA was made for web bots and is therefore more suitable than some of the other libraries, having support for specific Selenium-specific problems.

Functionality	Package								
	HMM[a]	***PyClick***[b]	***BezMouse***[c]	***pyHM***[d]	***Scroller***[e]	***ClickBot***[f]	***Typing***[g]	***[Noo19]***	***HLISA***
Mouse movement functionality									
Realistic mouse movement speed		✓	✓	✓		✓			✓
Movement accelerates/decellerates		✓	✓	✓					✓
Movement shivering		✓		?[h]					✓
Curve in movement[ij]	✓	✓	✓	?[h]		✓			✓
Moves to random location in element[i]									✓
Might overshoot the target			✓						
Acounts for target size									
Acounts for target distance									
Acounts for direction of movement									
Click functionality									
Realistic dwell time[i]				?[h]		✓			✓
Realistic flight time[i]		✓	✓			✓			✓
Simulates accidental right click			✓						
Simulates accidental double click			✓						
Simulates accidental no click			✓						
Simulates accidental misclick									
Scrolling functionality									
Pause between scroll ticks					✓				✓
Pause for finger replacement					✓				✓
Realistic scroll distance in tick					✓				✓
Supports Firefox's smooth scrolling									
Keyboard functionality									
Flight time[i]							✓	✓	✓
Dwell time[i]									✓
Simulates pauses in longer texts								✓	
Simulates typing errors									
Special interaction in password fields									
Correct usage of modifier keys									✓
Other features									
Selenium ready	✓				✓				✓
Accounts for visibility of elements		n/a	n/a	n/a		n/a	n/a	n/a	✓
Data based on experiments or sources			✓		✓			✓	✓

Table C.1: An overview of packages and functionality. A '✓' indicates the functionality is present in the package or code piece.

[a] "Human-like mouse movements": code piece to use B-splines `https://stackoverflow.com/a/48690652`
[b] Python package to generate human mouse movements `https://github.com/patrikoss/pyclick`
[c] Python code to simulate human mouse movements `https://github.com/vincentbavitz/bezmouse`
[d] Python package to imitate human mouse movements `https://pypi.org/project/pyHM/`
[e] Python package to simulate human scrolling `https://github.com/hayj/Scroller`
[f] Java code for an autoclicker `https://github.com/amSangi/ClickBot/`
[g] "type like a real person": code piece for typing `https://stackoverflow.com/a/15238748`
[h] The project links to source code that is incomplete
[i] Absence of this feature triggers interaction based bot detection in a simple interaction based bot detector
[j] Previously reported to be required to bypass Google reCaptcha `https://stackoverflow.com/a/37220168`