Test Bench for Data Observatory

Shivam Choudhary[1], Wenyu Zhang[2]

Abstract
Data Observatory is a tool used to analyze DOM trees of web-pages. OpenWPM[2] is a flexible web privacy measurement platform, which can help collect and annotate the web-page source by crawling through them. By integrating Test Bench and Data Observatory, we can find limitations of Data Observatory. This would enable us to use it more judiciously and would help expand and improve the detection process.In order to test the accuracy of Data Observatory, we developed a Test Bench System to test its detection. Since the system is controlled by us we can backtrack the results and can use it to find the accuracy of the detection process. Our implementation of the Test Bench runs as a Django based web application, and aims to bring together OpenWPM, Test Bench and Data Observatory using many wrappers.Finally we consider the performance of Data Observatory by evaluating the results.

Keywords
Test Bench; Data Observatory; P-values; OpenWPM

[1] *Electrical Engineering,Columbia University*
[2] *Electrical Engineering, Columbia University*

Contents

Introduction

The web privacy measurement issue has become an extreme important topic in the privacy area. Different web service providers are using users' information to customize their websites. So quantifying factors impacting websites' behaviors is the key to address this problem.

There are some tools that perform well in web privacy measurement. OpenWPM[2] is one such implementation by faculties/students from Princeton University. It is a flexible, modular web privacy measurement platform that can handle privacy experiments. And it has already been used as the basis of several published studies on web privacy and security.

Data Observatory is a tool developed by Yannis Spiliopoulos at the Computer Science Department in Columbia University. It can analyze the DOM tree of websites and extract their customized content. However, since we don't have any information of how much customized content was shown on the web pages, it is hard to measure the performance of this system.

In this project, we designed and implemented a Test Bench that integrates OpenWPM and Data Observatory to make them work in unison. The major contributions of this work is a simple,powerful and unified interface for measuring the performance of Data Observatory.

Our test bench has provided useful performance measurement result of integrated Data Observatory and OpenWPM, and the result is discussed. The result provides us with visibility as to how the Data Observatory can be further improved and also discusses some limits of the system.

1. Components

The system consists of the following components

1. Data Observatory[3]
2. OpenWPM
3. Test Bench [4]

1.1 Data Observatory

Data Observatory can be loosely defined as a statistical correlation system to verify a certain hypothesis using P-values.

In its true implementation sense Data Observatory tries to correlate the DOM (page source) differences in the content shown to users when one/some of their attributes (like location from which they are visiting, time of visit, gender) change. A typical use case can be the differences in price, amenities shown to a same user when visiting the website from different locations[1].

In our analysis we have limited ourselves to the evaluation of Data Observatory and not using it to find websites which serve different contents to the users. Our study aims to find quantifiable limits (number of iterations, accuracy and limits of detection) for the Data Observatory.

1.2 OpenWPM

OpenWPM is an automated web privacy measurement tool. In essence it's a web crawler based on Selenium and uses Firefox as its web driver. Along with the DOMS of the web-pages it also tracks the cookies that the website stores.

In Data Observatory, OpenWPM is used as an **aggregation** tool to collect any (specifically Test Bench) website DOMs. While running the tests, Data Observatory instructs the OpenWPM to annotate each tag of the HTML with **data-observatory-position** tag which the Data Observatory uses internally while running the evaluations.

1.3 Test Bench

Test Bench is an automated framework to evaluate Data Observatory.It is based on Python and uses Django as a web application framework. It interfaces with OpenWPM for DOM collection. Each web page that is being generated depends on the previous interaction of the user (in a probabilistic manner). We use attributes(defined in Semantics) to find the pre-configured probability associated with each attribute and serve content based on that.

Since we know which of the content was customized in advance we can back track the detection using the p-values generated by Data Observatory.

2. Semantics

We are defining Definitions of all the components/ terms that we have used throughout the experiment

Attributes We define Attributes as any personal identifier that can have a different value for different user, set of users from different countries, users of a particular age group and so on.

Probability Probability is associated with an attribute and is defined to be the number of times a particular content appears out of n iterations.

Experiment Experiment constitutes a full set of iterations, with **fixed probability** and attributes.

Corrected P-value This value is not defined by us but rather by Data Observatory [3]. We use this value element wise to ascertain whether it was detected or not. In the rest of the report P-value denotes Corrected P-value unless stated otherwise.

Threshold Across all the experiments the threshold is defined to be a fixed value of 10^{-3}. While we did have some discussion during the presentation that it should have been 10^{-2}, but since we were running short on time we could not re-run the tests (they do take a very long time.)

Success We define success for each iteration of an experiment in statistical sense. For a singly varying element we have
$p = probability\ of\ occurrence\ of\ an\ element$
$n = number\ of\ iterations\ for\ the\ set$
$N_{detected} = number\ of\ elements_{(P-Value<threshold)}$
$Thus, N_{expected} = p * n$
$Where\ N_{expected}$ =number of elements that should have been detected.

For Example, Lets say probability of appearance of an element is 0.9 and we are running 100 iterations for the subset of experiment. Since only single values are varied then we can be sure that at most 90 elements divs, DOMS or otherwise would have P-Values less than threshold.

3. System Architecture

Our system architecture is 3-legged, with Test Bench sitting in the middle. We use test bench to supply Open WPM with probability based DOMS and then after the data collection is complete we offload the detection to Data Observatory.

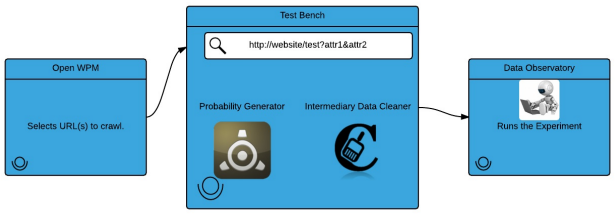

Figure 1. System Architecture

3.1 Overview

As is evident from the Figure 1, OpenWPM supplies the URLS to the Test Bench. Test Bench supports the URL in form of standard URL encoded query parameters. A typical example of query being supplied to Test Bench would be

```
www.testbench.com/test/?gender=
    male&country=germany
```

There is no limit on the number of parameters the system can take. The probability generator interfaces with the configuration file (see configuration in this section) for the desired experiment and renders the DOM for OpenWPM. These web-pages are then stored and asynchronously picked up by Data Observatory for detection.

3.2 Wrappers

Our wrappers are connectors which act as bridge between OpenWPM–Test Bench–Data Observatory. They are required because OpenWPM and Data Observatory don't inherently support interface access. Also since the experiments can be configured from the website, for quality of experiment (see quality in this section), we keep the data separately across the iterations and across the experiments.The

wrappers provide a way to set the iterations/number of browsers in an experiment and supply different probability across experiments.

3.3 Data Observatory Dictionary Generator

Data Observatory needs some reference of location. So the OpenWPM wrapper modifies the name of the file and appends it with the query-parameter and after its done collecting the data from the website it iterates over the directory which has the generated results and generates mappings to location for each file. Below is a part of JSON format that we create for the mappings. This JSON has a key "all_keys" in which all the geographical locations can be specified.Also one should keep in mind that if this file is not supplied to Data Observatory it will not work correctly. This is not exactly innovation or any contribution from our side, but we are including this for completeness of the report.

```
{
   "7_9_localhostcountry=india&
    gender=female.html": {
      "geo": "india"
   },
   "all_values": {
      "geo": [
         "germany",
         "india"
      ]
   }
}
```

3.4 Configuration

Our configuration file is a JSON key-value dictionary shown below. This file is customized per experiment and is kept constant throughout the experiment. New keys and their corresponding values can be easily added and the dictionary can be extended to support many other attributes.

```
   {
       "country":{
          "germany":{
             "0.1":"German",
             "0.9":"Default"
          },
          "india":{
             "0.5":"Indian",
             "0.5":"Default"
          }
       },
      "gender":{
          "male":{
             "0.7":"Male",
             "0.3":"Static"
          },
          "female":{
             "0.8":"Female",
             "0.2":"Static"
          }
       }

   }
```

Config Structure The top level elements specify the keys which are the same in the query-params,the second level correspond to the value of the attribute. We maintain a running counter which generates a random number N(0,1) and updates the counter. This is to ensure the probabilities depend on previous interactions with the system.(see Self-Evaluation for a discussion on probability generations)

3.5 Data Observatory Wrapper

The Test bench wrapper interacts with the Data Observatory and does the following:-

- Extracts only the elements which have corrected P-values.(See Appendix for code snippet)
- Cleans the Data Observatory intermediate files.(See Quality in next subsection for why its important)
- Extracts the P-values and stores it element wise.
- Stores the results (success/failure) iteration wise for each experiment.

3.6 Quality

To ensure each iteration has no bias taken from previous interactions, the Test Bench – Data Observatory deletes all the intermediary values. The Data Observatory uses the generated values as a cache for experiments. This is a small bug in the implementation system which can severely skew the P-values. This makes the Data Observatory consider the previous experiment/iteration P-values and sometimes leads to erroneous detection.

Apart from that we introduce our own name space format in Data Observatory for storing the result of the experiments. Again we are introducing this only for completeness and to make sure tests are conducted in a clean environment.

4. Methodology

With all the basic components in place we can now introduce how we actually conduct the experiments. This section first discusses about the self-evaluation of generated probabilities and iterations. And then discusses about the method we used for for elements testing. The scope of our tests was limited to varying at maximum 2 elements in a DOM independently.

4.1 Self-Evaluation

Our goal was to find the minimum number of iterations we must run so that the probabilities that are being generated are stable. This is important because if we cannot guarantee from Test Bench side that the probabilities generated would be constant and reproducible we cannot guarantee whether they would be detected from Data Observatory side.

So we decoupled our probability generator and then ran it under decoupled conditions for 10.

More discussion on the minimum iterations required is discussed in Section Result.(see Appendix for probability generator code snippet)

4.2 Single Element Variation

All the content in the website was fixed and only one attribute (country) was varied.

Iterations: In OpenWPM the crawling can be done wither sequentially or using multiple browsers.

Probability	Iterations(Maximum)
0.9	288
0.8	242
0.7	288
0.6	288
0.5	288
0.4	450
0.1	800

Table 1. Maximum Running Iterations for Probabilities

Each iteration can be then defined as a tuple of (browsers, iterations), hence we defined our iterations as (15, 15) or (12, 12) and so on. Since we were using two URLS for correlations hence the P-values are generated over all the iterations hence (12, 12) becomes 144*2=288 iterations. In hindsight we could have defined it to some whole number but from an experimental point of view it hardly makes any difference.

Secondly the number of iterations might seem different for some of the probability. It's because once the Data Observatory converges (achieves 100% detection for a single element), we stop the iteration.

OpenWPM crashes after 40*40 browsers and iterations hence that's the maximum number of iterations we ran for probability of 0.1. More discussion is included in Section Result.

4.3 Two Elements Variation

In this experiment we varied two attribute and changed the probability. In some experiments, the probability was kept equal between elements while in some cases we varied them independently (here independence implies probability of one element was not equal to other attribute).

5. Web System Implementation

5.1 Overview

We used Django as the web application framework for our project. Django is an open source web application framework, written in Python. The main

reason that we chose Django as the web application framework is because both OpenWPM and Data Observatory are written in Python. So using a Python web application framework helps make the whole integrated system to be unified and easy to manage.

On the side of front end, we used JQuery library and Bootstrap to build the system web pages.

Every time when a user wants to display the chart of experiment result, the front end system sends an AJAX call to the back end system to get JSON formatted chat information, and shows that on the web pages.

The architecture of the web application is shown in the Figure 2:

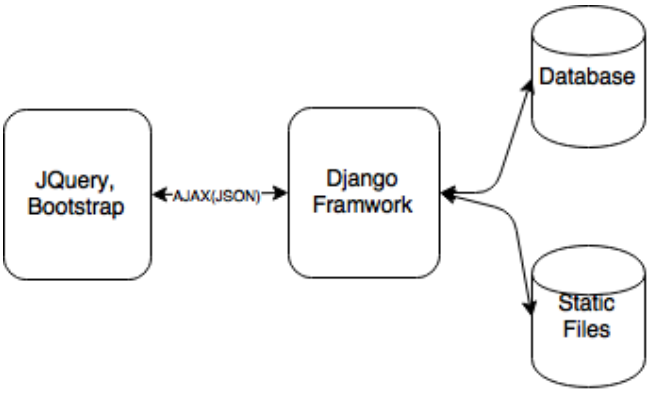

Figure 2. Web Application Architecture

In our test bench application, we created several pages to show various testing settings. From the screen shot of the web system below, we can see the option of “Dry Run Test”, “Single Element Page”, “Multiple Elements Page”, “Test Page”, “Configure Page” and “Visualization”. We will introduce the function of these options in the following sections.

5.2 Dry Run Test

In the Figure 3, we display the appearance probability of each element in DOM. If the line is relatively horizontal and becomes stable when the running times becomes long, it means we succeed in generating the DOM elements with the configuration file formatted in JSON. The web page also shows the corresponding testing result of this experiment result on the right side. All data is retrieved from the back end server. We keep the test result in folders named in their IDs of experiments.

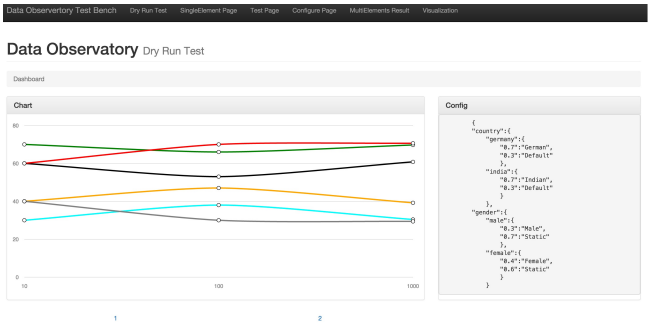

Figure 3. Dry Run Test Page

5.3 Single/Multiple Elements Test

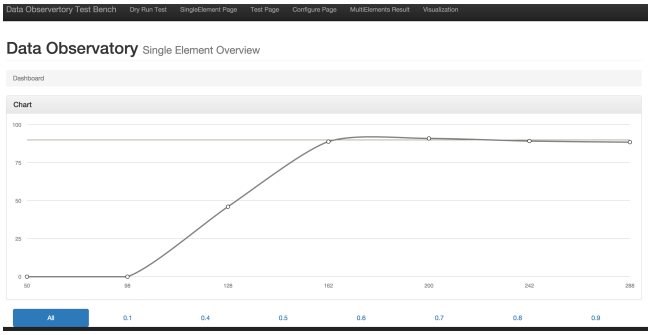

Figure 4. Single Element Test

In both single and multiple elements test pages, we use Morris chart to show the detected success rate. The lighter horizontal straight line is used to show the target success rate and the x axis represents the number of iterations we ran the experiment. The experiment result data is saved in the back-end database with their corresponding probabilities.

5.4 Test Page

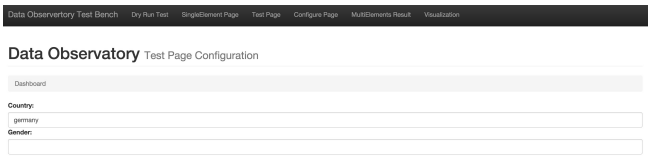

Figure 5. Test Page Generation Configuration

In the Figure 5, the web application requires a user to enter the current country and other information, which will be embedded in the query parameters of test URL. We basically copy the information from the table form and construct a corresponding query URL to ask for a dynamic generated web page.

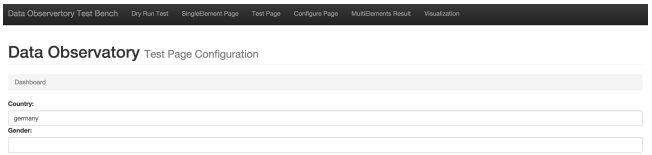

Figure 6. Test Page

At the back-end system, we made a template in Django to generate the dynamic web-page. The template fills the pattern with elements according to the setting in configuration JSON file. We put some static files (text, photos) in the static fold of Django so that the system can fill them in the place of pattern.

5.5 Configuration Page

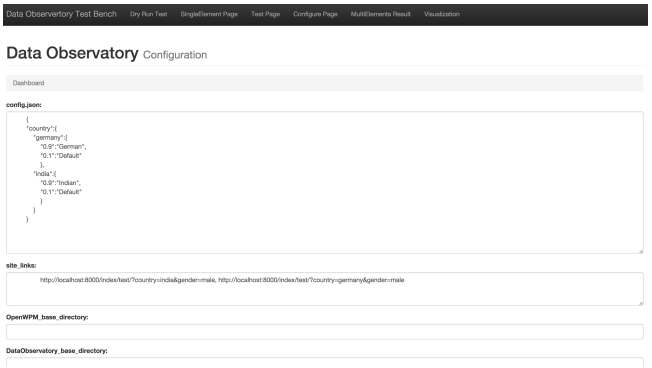

Figure 7. Test Bench Configuration Page

On the configuration page (Figure 7), user can change the current probability configuration JSON. Since our script used to run experiment needs to specify the directory of OpenWPM and Data Observatory, we can give the path to these directories on this page. We can also specify the experiment running iterations and output result file names on this page.

5.6 Visualization

The visualization page is used to show the result web page with annotations. This visualization web page was implemented by the teaching assistant of the course, Yannis Spiliopoulos. And we embed the visualization part into our system for the convenience of displaying experiment results.

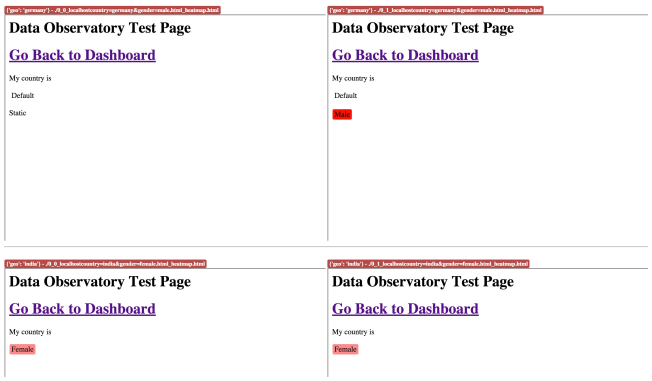

Figure 8. Visualization Page

6. Result

While using the above discussed methodology we got the following results.

6.1 Self-Evaluation

We ran the self-evaluation for a number of iterations and found that the results on an average converge around $10 \sim 25$ iterations hence the minimum number of iterations we should run for Data Observatory should be 25. Since we had six varying elements ('Default', German', 'Indian', 'Static', 'Female', 'Male') we ran the tests in one go for all the elements and counted the number of times Test Bench generated the element. Y axis specifies percent of time the element was generated and X axis denotes the number of iterations we ran for that particular element.Finally Figure 9 looks consistent with our expectations.Each title of a subplot shows the name and probability of element.

6.2 Single Element

For single element per Experiment we ran sets of iterations and calculated the number of elements having P-values less than the threshold.

Table 2 gives an idea of the number of iterations we ran for each experiment.

In Figure 10,Y axis denotes the percentage of elements which had P-values which had less than the threshold and the Title of each graph denotes the maximum percentage of elements allowed to have that value.Also X axis denotes the number of iterations that we ran for that particular experiment. To illustrate further let's say we run an experiment

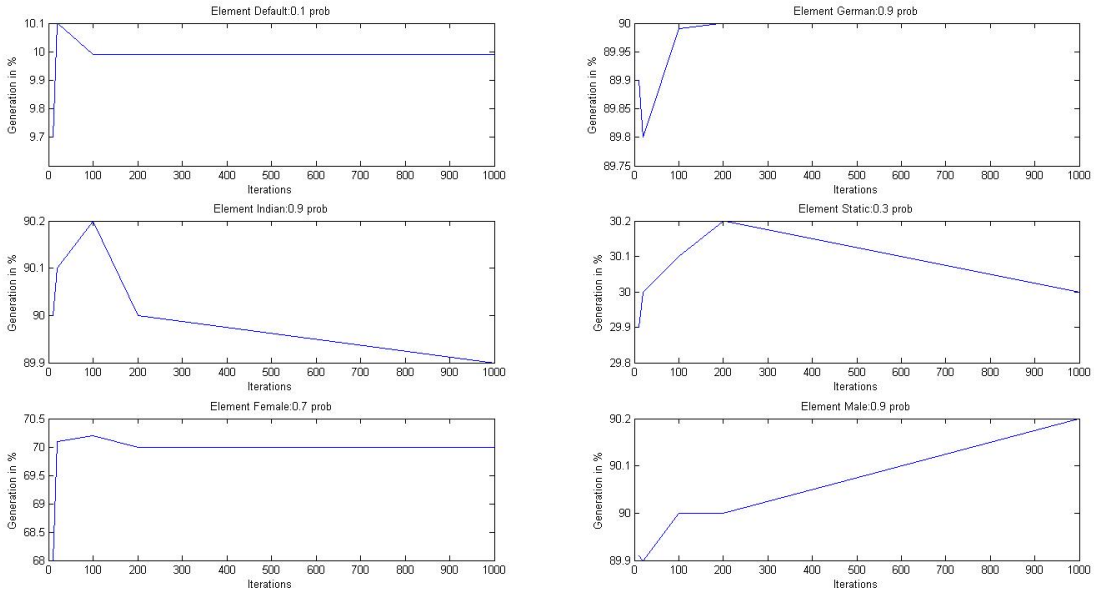

Figure 9. Self Evaluation

with 0.9 then in 100 iterations that element should have P value less than threshold at max 90 times.

Furthermore discussing on the results we can see that even in case of high probability (0.9), we must run at-least 200 iterations to be sure that the element can be detected. The detection becomes more difficult at lower probabilities and it reverses at 0.5. As is evident from graph it crosses the maximum detection range of 50 % and start detecting elements which were not supposed to be detected. Also if the probability of appearance of element is less than 0.4 it fails even after 800 iterations.

Probability	Maximum Iterations
0.9	288
0.8	288
0.7	288
0.6	288
0.5	288
0.4	800

Table 2. Iterations

6.3 Two Elements

Interestingly detection becomes more erroneous and tricky when Data Observatory is presented with elements that vary with different individual probabilities. We believe these results require individual attention per experiment so unlike single element, we will be showing each experiment result separately.

Case 1 Table 3 shows the first graph where the elements have the listed probabilities. As we can see from the graph some of the elements which were for the same number of iterations being detected easily have dropped down significantly in their detection percentage.

Table 3 shows the detection percentage for each of the element. Interestingly some of the elements (static, default) have not at all been detected although probabilities were low. 11

Element	Probability
Indian	0.9
German	0.9
Female	0.9
Male	0.9
Default	0
Static	0

Table 3. Case 1 Configuration

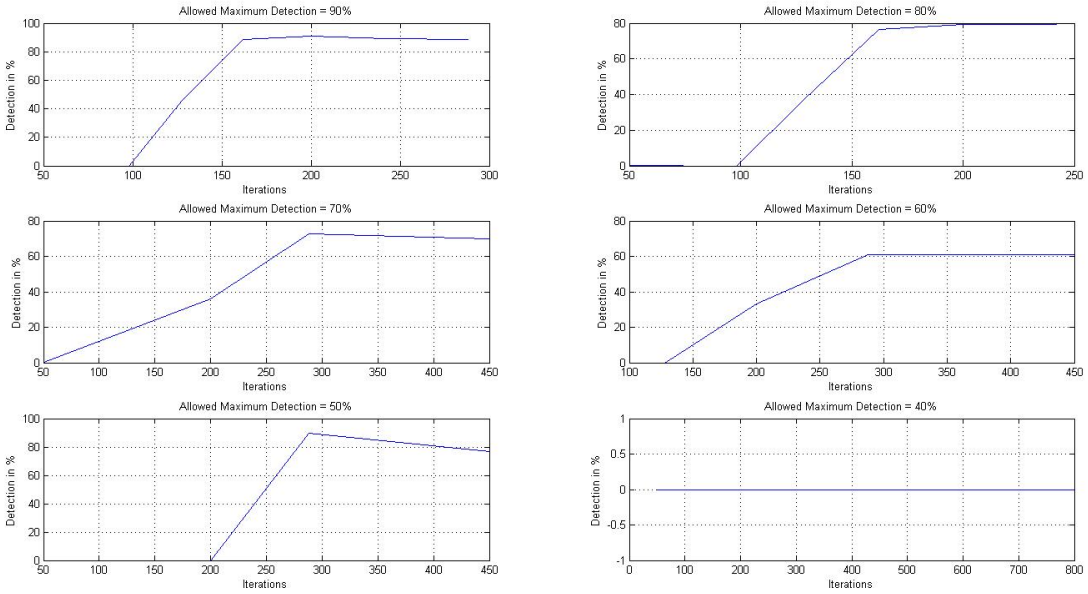

Figure 10. Single Element Result

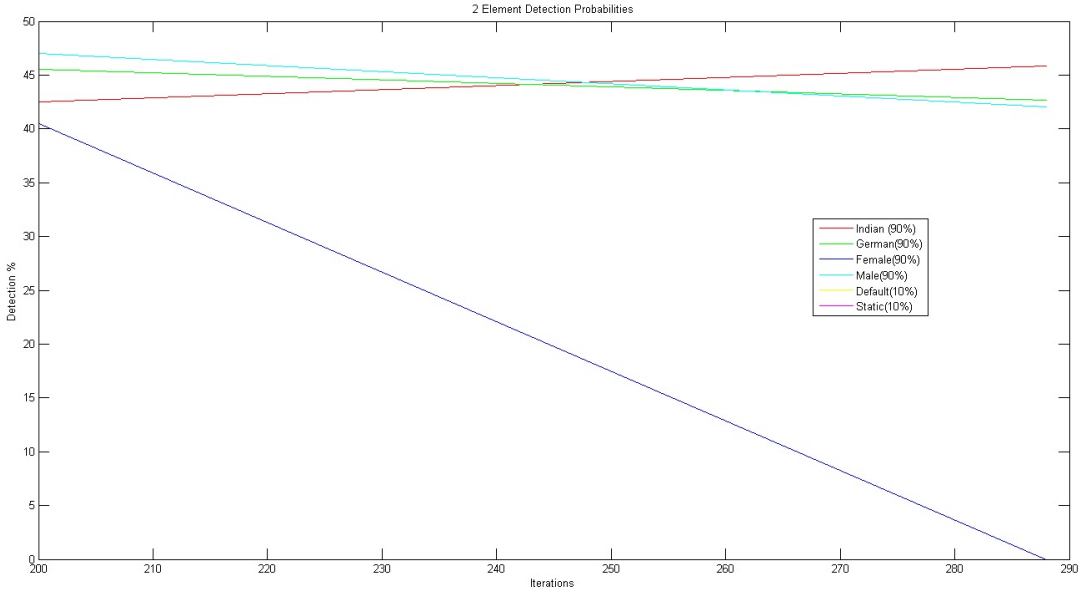

Figure 11. Case 1 Results

Obvious anomaly is with Element "Female" which was detected in First iteration but was missing in the second Iteration. Interestingly it did not appear even once whereas other elements like "Male" had a small dip in probability when iteration was increased.

Case 2 In this case we consider the elements with different probabilities as shown in Table 5. As is evident from the table that few elements have lower and different probabilities from each other. Table 6 shows the tabulated results for case 2.

Figure 12 shows the detection percentage graph for each element. When we vary the element probability differently across attributes we can see that

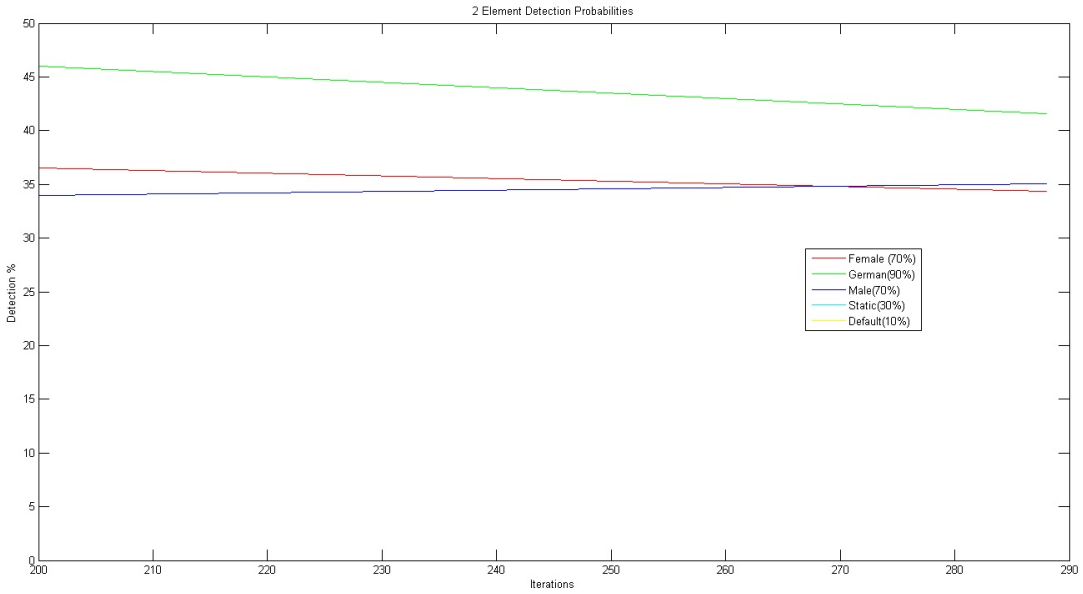

Figure 12. Case 2 Results

Element	Iteration 200	Iteration 288
Indian	42.5	42.7
German	45.5	45.83
Female	40.5	0
Male	47.0	42.01
Default	0	0
Static	0	0

Table 4. Results for Case 1

Element	Iteration 200	Iteration 288
Female	36.5	34.375
German	46	41.6
Male	34	35.06
Static	0	0
Default	0	0

Table 6. Results Case 2

Element	Probability
Female	0.7
German	0.9
Male	0.7
Static	0.3
Default	0.1

Table 5. Case 2 Configuration

the results are more or less consistent among the iterations. Two elements “Default” and “Static” were not detected in this case. This also does not show degradation in detection performance if the number of iterations are increased.

7. Conclusions

7.1 Single Element

For single attribute variation we would like to conclude that the system will not be able to differentiate if its probability of appearance is less than 0.4. Also at 0.5 we get abnormal detection. So the element that is appearing frequently in the website should appear more than 50% on an average to be detected. Also we would like to add that in practical scenarios the content that is shown to users from different locations (taking example of Booking.com) are so different that it would always detect, whereas if elements are varied with less probability detection might not happen.

7.2 Multiple Elements

For multiple elements, Data Observatory for the same number of iterations and all other parameters (attribute wise) remaining the same is not able to detect all the elements. We see a drop on average of 50%. Furthermore we have seen cases in which the Data Observatory stops detecting a particular element, again this can be attributed to the fact that the element's appearance probability is very low (0.4). So if a website varies lots of content each with different probability analogy being using more attributes to personalize the content Data Observatory might miss some of the elements. Also contrary to our expectations in some cases increasing the number of iterations reduces the probability of detection. We were not able to find any reason for this behavior even after running multiple iterations on same/under same conditions data set.

Acknowledgments

We would like to thank the teaching assistant of this course, Yannis Spiliopoulos, for his great help on understanding Data Observatory and OpenWPM, and his advice on system design of this test bench. We would also like to thank Professor Roxana Geambasu for her teaching us the knowledge on the course and advice on designing this test bench system.

Contributions

Shivam Choudhary: Designed OpenWPM/Data Observatory Wrapper API; Designed the success-based experiment; Designed Elements-Evaluation testing experiment; Helped in designing OpenWPM/Data Observatory integration system structure.

Wenyu Zhang: Implemented Django web application system; Designed test scripts and DOM generation scripts; Designed Self-Evaluation testing experiment; Helped in designing OpenWPM/Data Observatory integration system structure and Wrapper API.

Appendix

Probability generator This snippet is the probability generator which takes into account the previous interactions with the system using the SumK variable.

Listing 1. probability generator

```
def make_dom(dict):
    attr_dict = {}
    base_attributes =
        Common.read_config()
    config = Common.open_config()
    for k,v in dict.iteritems():
        if k in base_attributes:
            p = random.random()
            sumK = 0
            attr_values = config[k][v]
            for k1 , val1 in
                attr_values.iteritems():
                sumK += float(k1)
                if (p<sumK):
                    attr_dict[k] = val1
                    break
    return attr_dict
```

diff.py code changes We modified the diff.py code to extract the elements which have P-values less than 0.001. This snippet extracts those elements and puts them in a name-space file(code not shown here)

Listing 2. Diff.py changes for generating files

```
if correct_pvalues[element_id] is
    not None:
   pvalf[idx][element] = {}
   for k,v in
       correct_pvalues[element_id].items():
      pvalf[idx][element]['geo'] =k
      pvalf[idx][element]['pvalues']
          = v
      if v < 0.001:
        pvalf[idx][element]['detected']=True
      try:
        counter[element] +=1
      except Exception as e:
        counter[element] = 0
```

File Structure Below shows the generated JSON

file structure that we use. This shows that along with each element we capture its P-value and calculate the counts as well. File is truncated for brevity


```
{
 "0": {
   " Female": {
      "geo": "geo_india",
      "detected": true,
      "pvalues": 0
        .00014983087028584553
   }
 },
 "1": {},
 "2": {
   " Male": {
      "geo": "geo_germany",
      "detected": true,
      "pvalues": 0
        .00010966471321926009
   },
   " Default": {
      "geo": "geo_germany",
      "pvalues": 0
        .022174198597158863
   }
 },
 "3": {
   " Default": {
      "geo": "geo_germany",
      "pvalues": 0
        .022174198597158863
   },
   " Female": {
      "geo": "geo_india",
      "detected": true,
      "pvalues": 0
        .00014983087028584553
   }
 },
 "4": {
   " Male": {
      "geo": "geo_germany",
      "detected": true,
      "pvalues": 0
        .00010966471321926009
   }
 },
 "detected_counter": {
   " Male": 34.0,
   " Female": 35.5
 }
}
```


References

[1] Yannis Spiliopoulos, Differences on prices shown to users from different countries, http://www.cs.columbia.edu/ỹannis/stable/booking_com_us_ger_LA_feb01feb02_exp

[2] Englehardt, S., Eubank, C., Zimmerman, P., Reisman, D., & Narayanan, A. . *OpenWPM: An Automated Platform for Web Privacy Measurement*.

[3] Yannis Spiliopoulos, https://github.com/columbia/data-observatory/blob/master/dom_tools/diff.py

[4] Shivam, Wenyu https://github.com/shivamchoudhary/e6998