**Analysis of Website Consistency and Implications Regarding Domain Trustworthiness**

**A Senior Project in Computer Science for Seth Lifland**

**Yale University**

Seth Lifland

April 28th, 2015

Faculty Advisor: Bryan Ford

Key Collaborator and Graduate Advisor: Daniel Jackowitz

Director of Undergraduate Studies in Computer Science: James Aspnes

Yale University

**Abstract**

This analysis looks at website consistency as a potential metric for website trustworthiness. We define website consistency as the degree to which a website is formed from identical resources across multiple fetches of the website. Every Internet user has had the experience of typing in the same webpage on multiple occasions and seeing different contents. However there is more to website consistency than the pure visual appearance of a webpage; it is possible for a website to refer to the same resource with multiple URLs, thereby returning an image or a widget that looks identical to the end user while simultaneously encoding extraneous identifying information in the URL itself. We define a perfectly consistent website as one such that upon fetching the website several times sequentially, every resource referred to by the top level URL is identical and maps to identical contents across all fetches.

Our input data set is the Alexa Top 100 International sites list. We initially fetch each site several times using a scriptable browser and for each fetch, store a list of resources requested by the top-level site as well as the hashes of their contents. A processing script then analyzes the results of all fetches to construct observations about the consistency of the website that can be compared with other sites. Additionally, in the case of slightly-varying resource URLs that map to the same contents the processing script attempts to reduce this set of “synonym” URLs to a single URL with only the parts that are common to all variations and determines whether fetching the “reduced” URL yields the same resource as any URL in the original “synonym set.”

Our preliminary results indicate that website consistency varies widely across sites in our data set. Additionally, the number of synonym URLs associated with each site and the success rate of reducing them to a single URL and acquiring a resource with the same contents varies widely. Future work is needed to adequately account for these variations, although we offer some conjectures as to their possible causes.

**I. Motivation**

Anyone who has used the Internet in the last 20 years is likely aware of the fact that some websites are more trustworthy than others. Examples of “untrustworthy” behavior range from tracking users through the use of cookies or fingerprinting to attempting to trick the visitor into installing malicious software locally on her machine. Trustworthiness in the case of websites is a difficult quality to define, and it is equally difficult to prove trustworthiness empirically. However, there are certain website behaviors that we can identify and compare across websites to make conjectures about which websites are more or less likely to be trustworthy.

This analysis offers a potential first step towards measuring website trustworthiness by looking specifically at website consistency. We define website consistency as the degree to which a website is formed from the same parts across all fetches. Specifically, a top level domain generally requests a list of other resources each time it is fetched. For our input data set – the Alexa top 100 International sites list – we are interested in which and how many of these resources are consistent or somehow inconsistent across all fetches of a given website. Eventually, we hope to be able to use these data to form conclusions about the relative trustworthiness of various websites.

In addition to evaluating website trustworthiness, this analysis is also interested in evaluating methods of combatting untrustworthy behavior. One form of potentially untrustworthy behavior which the analysis is interested in combatting is user-tracking through information embedded into the URLs of resources themselves. For instance, a long query string in a URL might contain one element of the form “key=value” whose value has no effect on the resource retrieved, but which is being maintained only to identify the user. The analysis attempts to identify this behavior by looking for sets of “synonym” URLs in the lists of resources for all fetches. A group of URLs are considered synonyms if they vary slightly in structure but return resources whose contents hash to the same value. Based on the assumption that some of the information that varies between otherwise identical URLs which return identical resources could be used to track the user, this analysis attempts to strip out the unique parts of synonym URLs and determines whether the “reduced” URL is able to return the same resource as all of the synonym URLs. In cases where fetching the “reduced” URL successfully returns the same resource as any one of the original synonym URLs, we can feel secure in that we are no longer cooperating with the website’s attempts to track us as a user. Ultimately, we would ideally like to be able to take any URL that a website requests, evaluate elements that are likely extraneous and potentially used for tracking, and remove them before proceeding with the request.

**II. Implementation**

**II.1 Organization**

The end-to-end analysis system consists of several types of components. The detailed processing scripts are implemented in Python, the actual fetching of websites and writing of results is done by a pair of JavaScript scripts using the SlimerJS scriptable browser, and the easy processing of all the input sites and management of output files is handled by several bash shell scripts. An illustration of the system with all the source files named is provided below:

**sites**

**map.sh**

**results**

**dprocess.sh**

**process.py**

**survey.js**

**<Other python modules>**

**fetchsyn.js**

**resultstatss**

Figure 1: High-level organization of the website consistency analysis system

The following is a high-level description of each of these components.

* sites contains several possible lists of websites to evaluate, with just the top-level domain names printed one-per-line. For all of my experiments, I used the Alexa Top 100 International list.
* map.sh is responsible for taking every website in a given text file, feeding them to the survey.js script a given number of times, and storing the results in a strictly-organized directory that will be read later by dprocess.sh.
* survey.js uses SlimerJS to fetch a provided website and write the results to a provided file as a JSON object which can later be directly parsed by process.py.
* results contains the results of fetching all the sites a given number of times with map.sh. Within the results directory, map.sh creates a directory for domain name fetched, creates a directory 1-(num fetches) for each fetch, and writes the fetch result to a file called results.json.
* dprocess.sh (detailed-process) passes the results files for all fetches of all sites to process.py and ensures that the output is correctly stored in the resultstats directory.
* process.py is the core of the analysis program. It takes the list of results.json files corresponding to all fetches of a given site as input, computes various statistics on the resources present across all fetches, identifies “synonym URL sets,” computes and attempts to fetch “reduced URLs” for each synonym set, and outputs results both on a per-site basis to a directory within resultstats and to a couple of aggregate results files, stored in resultstats/agg.
* The other python modules that process.py refers to provide functionality such as reducing synonym URL sets to reduced URLs, fetching reduced URLs, and sorting resource URLs into various potentially interesting data structures. Currently, the most important of these modules are urltable.py and synurl.py.
* fetchsyn.js is almost identical to survey.js in that it simply fetches a URL and writes the received resources to a location where they can be processed later. Fetchsyn, however is invoked directly by process.py for fetching reduced URLs in order to determine whether they are valid substitutes for the synonym URLs from which they are derived.
* resultstats is where dprocess.sh stores the results of running process.py on all the top-level sites. There is a directory for each site inside resultstats, containing a text file describing the detailed results of executing process.py on that site. Additionally, the results of fetching the reduced URLs calculated for a site are stored in the site’s corresponding directory within resultstats so that they don’t have to be refetched by default every time the processing script is run.[[1]](#footnote-1) Resultstats also contains a directory “agg” (aggregate data) which contains text files and csv files that contain concise data from across processing all sites.

**II.2 Processing Details**

This section will describe in detail how process.py processes a single site. Process.py is run once per site and its input is the list of “results.json” files corresponding to each fetch of the site. These results files are directly interpreted as Python dictionaries. The analysis is most interested in the list of resources associated with each fetch, where a resource is represented by the URL of the resource received, the hash of its contents, and its size. It is on these resource lists that the vast majority of the processing acts. In order to facilitate efficient categorization of the resources, Process.py maintains several data structures as it processes the results.json file for each fetch of a site:

* a dictionary mapping a URL to the number of times it occurs across all fetches
* a dictionary mapping a URL to all possible hashes with which it is associated
* a dictionary mapping a hash to all possible URLs with which it is associated

Given these data structures, process can place each URL into one of the following categories:

* Consistent – same URL, same contents in all successful fetches
* Content-Inconsistent – same URL, contents not same in all fetches
* Synonym – different URL, same contents in all fetches
* Totally Inconsistent – URL and contents vary in all fetches
* Failed – URL was not fetched successfully

The categorization algorithm is implemented inside Process.py in the function “categorize\_ resources\_by\_fetch” and works like this:

For each resource list:

For each resource in the resource list:

If the size is 0:

resource -> failed resources

Else if resource hash is associated with multiple URLs:

resource -> synonym resources

Else if resource URL is associated with multiple hashes:

resource -> Content-inconsistent resources

Else:

If resource fetched appears in resource lists for all successful trials:

resource -> Consistent resources

Else:

resource -> Inconsistent resources

This algorithm is imperfect, but it maintains the invariant that no single resource URL is sorted into more than one category for a given fetch, although there can be some overlap between the categories. For instance, if there is some hash that multiple resource URLs return, these URLs could both appear consistently in the resource lists of all fetches, but they will still be identified as synonyms for the same hash, and therefore be categorized as such, despite also being consistent. One of the greatest difficulties of this analysis was conceiving of mutually exclusive categories for the different resource URLs that take into account their structure, the hashes of their contents, and their success.

One improvement to the analysis that I have not had time to implement is to attempt to associate each URL within a resource list with the *single most similar* URL in every other fetch. This would avoid the above situation, because the two URLs from the same fetch which map to the same contents would be placed into separate sets with only identical URLs as seen across all fetches, and therefore would be categorized as consistent, whereas a “true” synonym URL would correspond to a different set where all the URLs from different fetches are slightly different, but whose contents hash to the same value.

Once the URLs are sorted into these sets, the only one which we process additionally is the synonym URL set. The additional processing involves taking each set of synonym URLs and attempting to reduce it into a single URL where all the unique parts have been stripped out. For instance, often a set of synonym URLs will each contain a long query string with elements of the form “key=value” in which all values are identical except for one or two. In this case, the output of the reduction algorithm would be a URL whose query string has simply omitted the “key=value” segments for which the values differed.

Recall that the analysis is interested in evaluating the effectiveness of stripping the unique parts from synonym URLs down and fetching the reduced versions as a potential method of inhibiting website tracking through extraneous URL parameters. Therefore at this point Process.py attempts to fetch these reduced URLs. However, it first attempts to fetch one of the original URLs from the “synonym set” as a “sanity check.” We expect that fetching one of the original URLs with all the unique parameters left in should produce contents which hash to the same value as the original hash it was associated with. If the sanity check succeeds, Process.py attempts to fetch the reduced URL. It then compares the original hash to the hashes of all the resources retrieved by fetching the reduced URL, and if it finds a match, the reduced URL is deemed a successful substitute for any synonym URL in the set.

Process.py handles fetching in the case of the sanity check and reduced URLs by initiating a subordinate process and passing the URL to fetchsyn.js which attempts to fetch the URL, and writes the results of the fetch to another .json file which the process.py script can read from just as it did with the original results.json files. By default, when dprocess.sh is invoked, it will attempt to directly read the results of past reduced URL fetches from their expected locations in the file system because the fetch of all the sanity URLs and reduced URLs is extremely time-consuming and is not yet parallelized in any way.

There are four possible results when fetching a reduced URL:

* The fetch fails completely
* The sanity check fails, and therefore the reduced URL(s) associated with that synonym set is never tested
* The fetch succeeds, but no match for the original contents is found
* The fetch succeeds, and matching contents are retrieved

**II.3 The Synonym URL Reduction Algorithm**

The synonym URL reduction algorithm is probably the most complicated process in my code. At a high level, it involves splitting each URL in the synonym set into segments and constructing a reduced URL based on the “intersection” of the URLs. In my current algorithm, an “intersection” URL is calculated by incorporating each URL in the synonym set into the intersection URL incrementally. Each URL is first split into segments, where each segment is represented by a 3-tuple of the form (segment number, segment text, segment type) where possible values of segment type are scheme, network location, path, parameters, query, and fragment. These types are based on Python’s built-in urlparse library, which takes a url specified as a string and splits it into a 6-tuple with these components. The following is an illustration of a generic URL with each segment type illustrated:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| scheme | network loc. | path | params | query | fragment |
| https:// | example.com | /a/b/foo\_dir | ;x=3;id=24 | ?bar=a&y=52 | #cat\_pictures |

The algorithm further splits each URL component by the corresponding separator into a list of segments. For example, in the above URL, the “path” component would ultimately become the list [(3, “a”, PATH\_TY), (4, “b”, PATH\_TY), (5, “foo\_dir”, PATH\_TY)]. Including the segment numbers in the tuples was more necessary for another data structure which used the split\_url function defined in urltable.py because in that data structure the segments became dictionary keys corresponding to lists of possible segment text values for segments that had different possible values.

The input to the reduce\_url function is a set of synonym URLs represented by these lists of tuples. The algorithm proceeds in the following way:

intersection\_url <- synonym\_urls[0]

for each url in synonym\_urls:

\*\* check that url and intersection url are same length and similar in structure

for each corresponding segment in both URLs

if segment types and segment texts equal

continue

else

intersection\_url[seg][seg\_text] <- “”

In this way, after processing all URLs, the intersection URL reflects only the segments that are identical to all URLs in the set. The new intersection URL is a list of segments where some have a text value equal to “”. The reduced URL is constructed with a standard reconstruct\_url function in urltable.py that undoes the work of splitting the URL into segments, removes the empty segments, and assembles the result into a single reduced URL string.

The line marked with \*\* requires a bit of explanation. As it turns out, while synonym URLs in a set are often nearly identical sometimes synonym they can be different lengths or fundamentally differ in structure. Because my current algorithm depends on comparing synonym URLs that are of the same length and are close to identical, I have to treat synonym URLs that don’t “fit the same mold” separately. In actuality, the algorithm I use is slightly more complicated; a synonym set is actually reduced to a list of synonym URLs. Therefore if a URL does not pass the check in the \*\* line,

A different algorithm that I suspect might work better but have not had the chance to implement would involve looking segment-by-segment rather than URL by URL; that is, looking at the first segment of all URLs in the set at once and either picking the most popular segment if there’s a plurality or majority vote for one segment value, or simply omitting the segment or choosing at random if it differs across too many of the URLs in the set.

* Later sections:
  + Future Work
    - unanswered questions: why does sanity check fail?
    - Better similarity algorithm (order-invariant)
    - creating similarity sets for all resources where a set has no more than one URL per fetch – this is interesting because we can look at number of similarity sets vs synonym sets, etc.
    - Mention the graphs that he suggested I make
  + Challenges
    - Categorizing URLs cleanly; there tends to always be some overlap
    - The current categorization causes any resource which isn’t fetched successfully at least once but is otherwise consistent to be categorized as inconsistent – prevent this
    - Currently have to remove duplicate URLs from resource list for categorization code to work; if a URL occurs multiple times within a single fetch only its first occurrence will be preserved and processed
    - Note that fetch of some sites failed every time – sort of looks like lots of social networks and sites that require logging in; possibly failed because of bot-protections?
  + Synonym URLs can also be consistent, but they will be categorized first as synonym; if the same contents appears multiple times in a fetch referred to by two different names, the two names may appear consistently in all fetches, but they will still be counted as synonyms
* Interesting results
  + More sites with 100% consistent bytes than 100% consistent resources – likely because failed resources have no impact on %consistent resource bytes
  + Also important to note that some of the sites at the top of the consistency % charts are very small overall – eg. Baidu.com and youradexchange.com
  + Important to note changes in scale in non-% graphs
    - Highest values in the bottom 50 graph still lower than the lowest values in higher graph, scale is an order of magnitude different

1. It is possible to force dprocess.sh to refetch all synonym URLs for all sites by invoking it with the flag “-refetch” [↑](#footnote-ref-1)