UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

RELEVANCE AND PRIVACY IMPROVEMENTS TO THE YACY DECENTRALIZED WEB SEARCH ENGINE

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

JEREMY RAND
Norman, Oklahoma
2018

RELEVANCE AND PRIVACY IMPROVEMENTS TO THE YACY DECENTRALIZED WEB SEARCH ENGINE

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Dean Hougen, Chair

Dr. Qi Cheng

Dr. Sridhar Radhakrishnan

"...dedicated to all the people who are out there unfucking the Internet."
– Giordano Nanni
Historian, Journalist, Founder of The Juice Media, Co-Creator of Juice Rap News.
https://www.youtube.com/watch?v=5b5W_U8RBDI

Acknowledgements

We'd like to thank Ryan Castellucci and midnightmagic for some fascinating and productive discussions regarding anti-Sybil mechanisms.

Contents

Abstract

The YaCy decentralized web search engine carries significant potential advantages in censorship resistance over centralized search engines such as Google. However, YaCy currently suffers from deficiencies in relevance of results as well as weaknesses in privacy. We have developed improvements to YaCy's relevance, including tools to generate a ranking dataset that can be fed to machine learning algorithms, fixes for some significant YaCy flaws that severely damaged ranking, and tools for ensuring that the decentralized index contains relevant results. We have also conducted an initial privacy audit of YaCy's usage of anonymizing proxies and YaCy's application-layer protocol, with recommendations for improving YaCy's privacy in both areas. We believe that this work helps pave the way for YaCy to become a credible competitor to centralized search engines. We expect future work to experiment with various machine learning implementations using our ranking dataset generation toolset, as well as implementing the improvements recommended by our initial privacy audit and conducting more extensive privacy audits once our initial recommendations are implemented.

Chapter 1

Introduction to Decentralized Web Search

In this introductory chapter, we first provide background on search engines and their impact on civil liberties. We then examine the motivation for decentralized web search, from the perspectives of censorship resistance, privacy, decentralization, transparency, and software freedom. We close the introduction with a summary of our contribution to the field.

1.1 Background on Search Engines, Civil Liberties, and Decentralization

Today's society has reached "the age of cypherpunk", where civil liberties, human rights, and government policy are inextricably tied to code and the Internet. Free speech and privacy rights on the Internet (and, by extension, in all of society) are at risk from central authorities who seek to control information via censorship and wiretaps. While significant work has been accomplished on censorship-resistant and wiretap-resistant protocols of information exchange, such as Tor, there has been a relative lack of high-quality work in the field of web search engines (PRISM Break contributors, 2018). As has been shown by enormous documentation, Google (2018a), which has a large share of the web search market, is definitely tampering with search results and logging queries. Most of Google's competitors, such as Bing (Microsoft, 2018) and Yandex (2018), are not particularly better on these points. DuckDuckGo (2018b) is certainly an improvement in terms of pri-

vacy, since it is accessible via Tor onion service (The Tor Project, 2018c), but it has no way to prove that it is not tampering with results; this is inconsistent with the cypherpunk (Wikipedia contributors, 2018d) principle of removing trusted third parties wherever possible.

An alternative search engine methodology has been proposed by the YaCy project (Christen, 2017), which uses a peer-to-peer distributed hash table (DHT) (Wikipedia contributors, 2018e; Maymounkov & Mazires, 2002) to store a search index, with results being determined collectively by the network. YaCy states that a single user who is storing 10 million web pages in their YaCy index can expect to use around 20 GB of storage (YaCy Wiki contributors, 2014); for comparison, Google's search index currently contains hundreds of billions of web pages, and uses over 100 PB of storage (Google, 2018b).

YaCy has a major civil liberties advantage, in that it is significantly more difficult to censor results for YaCy than for centralized search engines. Unfortunately, YaCy's search ranking performs poorly compared to Google, Bing, and DuckDuckGo (Jordan et al., 2018), and YaCy's privacy is not particularly good. YaCy's poor ranking quality is particularly problematic from the perspective of censorship resistance, because even though YaCy is resistant to conventional censorship (i.e. deleting results from the index), poor ranking algorithms that hide relevant information in a sea of spam are themselves a form of de facto censorship (Masnick, 2018).

We argue that improving YaCy's ranking and privacy is a better approach for solving these issues than continuing to rely on centralized search engines; we have done preliminary work to this end.

1.2 Motivation for Decentralized Web Search

This section covers some of the flaws in currently existing centralized search engines, which we believe YaCy has the potential to fix. Note that the following concerns use Google as an example, but these concerns apply to all of Google's centralized competitors as well.

1.2.1 Censorship Resistance

In 2016, Google received over 1 billion URL removal requests due to claimed copyright infringement. 914 million were removed (Van der Sar, 2016). It is fairly obvious that no human reviewed those URL's before removing them from Google. This poses a serious risk for censorship. In addition to copyright-based censorship, Google has also begun engaging in political censorship (Sommer, 2017; Damon, 2017a; Damon, 2017b; Damon & North, 2017; Damon & Niemuth, 2017; Sweatte & Damon, 2017; Parry, 2017). Even if a centralized Google competitor claimed to not censor results, we would have no way to verify that this was the case.

1.2.2 Privacy

Google also tracks user searches for targeted advertising purposes. This poses a serious risk for privacy, particularly given that Google's data, as well as the data of its competitors Bing and Yahoo, is regularly collected by the U.S. government (National Security Agency et al., 2013). Even if a centralized Google competitor claimed to not log any user data, as is the case for StartPage (2018) and DuckDuckGo (2012), we would have no way to verify that this was the case.

1.2.3 Decentralization

Google is also centralized; this design is less secure against some failure modes than a decentralized system would be. For example, if, in the future, Google goes bankrupt or undergoes a hostile takeover, this would most likely result in disruption to the Google userbase. (The same issue applies to Google's centralized competitors.) Centralized systems that are politically relevant (which definitely applies to search engines that the general public use to stay informed) have also historically been magnets for lawsuits. For example, Flooz (Higgins, 2014) and Napster (Wikipedia contributors, 2017a) both failed due to legal troubles stemming from centralization, while their decentralized successors—Bitcoin (Nakamoto, 2009) and BitTorrent (Wikipedia contributors, 2018a), respectively—have fared much better legally.

1.2.4 Transparency

Google's ranking algorithm is a black box with no transparency or accountability. Research by Robert Epstein (2015; Camp, 2016) suggests that a subtle change in Google's ranking algorithms could influence and even swing elections. Amit Singhal, who was then the head of Google search ranking, responded to Epstein's research by saying that "our search results" "must be trusted to be considered valid", and that "we work very hard to earn and keep the trust of everyone" (Singhal, 2015). Google's response only serves to underscore the problem: because there is no way to audit what the Google ranking algorithm is doing, we are being asked to simply trust that Google is non-malicious and non-compromised. Eric Schmidt, a Google executive, also was a major supporter of Hillary Clinton's 2016 Presidential campaign (Podesta, 2014; Mills, 2014; Higgins, 2016). This poses a risk to free elections. While we are not aware of known conflicts of interest regarding Google's centralized competitors

and political candidates, there is still no way to audit that political or commercial manipulation is not happening.

1.2.5 Software Freedom

Google's software is not libre (Free Software Foundation, 2017b); this is problematic for users who would like to study or adapt the software. This concern applies to Google's centralized competitors as well, with the partial exception of DuckDuckGo, which is partially libre (DuckDuckGo, 2018a). However, even DuckDuckGo is not fully libre: while it is possible to contribute code to certain components of DuckDuckGo, it is not possible to fork the DuckDuckGo code and run your own instance.

1.3 Our Contributions to YaCy

This thesis covers our work in three areas:

1. Relevance improvements related to machine learning. These include tools for generating a ranking dataset that can be fed to machine learning algorithms, tools for simulating YaCy's ranking algorithms in a controlled environment, and a sample machine learning implementation that performs gradient descent learning against a generated dataset to produce better YaCy ranking.
2. Other relevance improvements. These include fixes for some significant YaCy flaws that severely damaged ranking, and tools to expand YaCy's index to include more relevant results.
3. Privacy improvements. These include partial audit results for both proxy leaks and application-layer privacy leaks in YaCy, and recommendations on improving both proxy-based privacy and application-layer privacy in YaCy.

Each of the above three topics is covered in a chapter.

Chapter 2

Related Work

This chapter briefly covers relevant pre-existing work in both the field of decentralized web search and the field of supervised learning.

2.1 Decentralized Web Search

YaCy (Christen, 2017) is currently the only libre (Free Software Foundation, 2017b) decentralized search engine that contains a crawler. Seeks (Benazera, 2014b) was a libre decentralized metasearch engine that utilized social search. Seeks identified peers with similar interests using locality-sensitive hashing rather than using explicit social graph information provided by the users. Seeks did not contain a crawler; its results were sourced from other search engines such as Google. Seeks is currently discontinued (Benazera, 2014a; Benazera, 2016). Searx (Tauber, 2018) is a newer libre metasearch engine. Like Seeks, Searx does not have its own crawler. Searx does not attempt to implement social search or any other P2P functionality. Blippex (2013b) was a partially-libre distributed search engine that utilized user behavior to crowdsource ranking information. The Blippex server was non-libre, but the client-side code (a browser extension) was libre. Blippex is currently discontinued (the links from their GitHub Pages blog are now dead). FAROO (Wikipedia contributors, 2017c) is a non-libre distributed search engine. Like YaCy, FAROO utilizes its users for crawling. FAROO utilizes user behavior for ranking, similarly to Blippex. Sciencenet (Lutjohann et al., 2011) is an adaptation of YaCy for searching scientific papers and data (particularly targeted at life sciences). Sciencenet has a much

simpler ranking problem to deal with compared to the public YaCy network, since the Sciencenet index consists of only curated academic papers and data, and is therefore likely to have little or no spam compared to the public YaCy network.

Since we are primarily interested in libre systems that include their own crawler, we find that YaCy is the only relevant decentralized search engine for our research.

2.2 Supervised Learning

In the field of machine learning, several common approaches to learning exist. Supervised learning (Caruana & Niculescu-Mizil, 2006) utilizes a pre-existing dataset of known "correct" input/output pairs, and tweaks the function being learned to conform to that dataset. Reinforcement learning (Michie, 1963) allows the function being learned to randomly wander, and gives positive or negative reinforcement for each change, allowing the function to conform to the feedback. Unsupervised learning (Barlow, 1989) allows the function to learn without any pre-existing datasets or reinforcement. When a dataset of correct input/output pairs is available, supervised learning will typically give better results.

However, most supervised learning algorithms are based on partial derivatives, and are therefore dependent on the learned function being differentiable (and having easily calculatable partial derivatives). While such a requirement is not feasible to achieve for non-libre search engines, since they generally only output a list of ordered URL's rather than a dump of their internal ranking function's execution, we observe that YaCy's libre nature makes it feasible to implement modifications that achieve this. As a result, we find that supervised learning is likely to be most applicable here.

Chapter 3

Learning-Related Relevance Improvements

In this chapter, we introduce "ranking transparency", a patch to YaCy that allows it to dump a machine-readable log of exactly what calculations led to its final ranking. We then discuss a set of Go libraries for handling ranking transparency dumps, including the ability to simulate how the ranking results would change with different ranking algorithms. We introduce a tool for generating a dataset of "correct" ranking choices, along with a design for a new web of trust construction that permits datasets from multiple users to be combined without leaking private data such as search queries, URL's visited, or social graph metadata, in a way that is resistant to spam and other malicious manipulation. Finally, we demonstrate our toolset with a simple example learning algorithm that uses gradient descent against a dataset to produce better YaCy ranking.

3.1 Ranking Transparency

YaCy utilizes two different ranking methods: Solr (The Apache Software Foundation, 2018b), an Apache project based on Apache Lucene (The Apache Software Foundation, 2018a), and RWI (Reverse Word Index), a custom YaCy-specific ranking function. The standard YaCy search interface consists of two steps: pre-ranking and post-ranking. Pre-ranking only takes individual documents into account, and can be performed using either Solr xor RWI. Post-ranking utilizes the set of document data generated during pre-ranking, and always uses RWI.

Both Solr and RWI use ranking rules that consist of numeric components. For

example, one component might increase a document's score if the search query appears in the document's title, and another component might increase a document's score if the document's URL is shorter. The numeric values of these components, called *boost values*, determine the magnitude of the effect. YaCy often advises users to tinker with the boost values, but due to the large dimensionality of the solution space, it is difficult to guess "good" values.

To perform supervised learning on YaCy's ranking, or to simulate its ranking algorithms in an external environment, it is necessary for YaCy to export information on how it calculated ranking scores, instead of simply providing a single number. We call this feature *ranking transparency*.

It should be noted that ranking transparency is highly beneficial even if we want to evaluate the ranking quality of a particular set of boost values without performing supervised learning, because YaCy's ranking implementation is not particularly fast. Simulating the ranking logic in custom, optimized code is substantially faster.

Solr has ranking transparency built-in. To enable transparency for all of the results which would normally be returned, specify the `debugQuery` parameter. To further enable transparency for a set of results which would not necessarily normally be returned, the `explainOther` parameter can be set to specify the set of affected results.

Unfortunately, YaCy does not provide a method of passing through the `debugQuery` or `explainOther` parameters when it calls Solr as part of a P2P search. Furthermore, RWI does not have any support for transparency. However, YaCy is capable of passing through those parameters to its local Solr instance, via the `/solr/select` endpoint.

While it is possible to perform supervised learning and ranking simulation using solely YaCy's local Solr instance, this is of limited usefulness, because both the local Solr search index and the Solr ranking algorithm are a subset of what YaCy

end users are exposed to. Instead, we elected to attempt to implement ranking transparency in YaCy.

As a first step, we modified the YaCy code that calls Solr in remote peers to enable `debugQuery` when a special setting is enabled in YaCy, and added the necessary plumbing to return that data to the user. Next, Solr by default only returns the ranking transparency data in XML format; we had to adapt a JSON serializer (used in Solr for other purposes) for Solr's ranking transparency data so that YaCy could return Solr ranking transparency data in JSON format like the rest of YaCy's output. Finally, the Solr ranking transparency data isn't sufficient on its own; we also wanted RWI ranking transparency data. We ended up writing some Java functions that could construct Solr-formatted ranking transparency dumps based on the RWI post-ranking calculations. We haven't yet implemented RWI pre-ranking transparency, which means that the only results that have full ranking transparency are the results that use Solr for pre-ranking.

The YaCy ranking transparency code is submitted as a pull request to YaCy (Rand, 2016d), but is not yet merged. We intend to get it merged in the future.

3.2 Transparency Dump Handling

This section covers the procedure used to obtain ranking transparency dumps in a format suitable for efficient processing. We cover the initial parsing procedure for the dumps that Solr/YaCy outputs, and then discuss optimization steps that are applied after initial parsing.

3.2.1 Parsing

A transparency dump, as outputted from Solr/YaCy in JSON format, follows a structure as specified by this Go `struct`:

```
type SolrJSONDump struct {
    Match       bool
    Value       float64
    Description string
    Details     []SolrJSONDump
}
```

`Match` is generally always `true` for the cases that we care about, i.e. the cases where a search result matched a query. If a result is retrieved via `explainOther` that does not match the query, `Match` will be false.

`Value` is the ranking score; higher scores are more relevant.

`Description` is where most of the interesting information exists; it consists of a mostly human-readable explanation of what calculations took place to provide `Value`. If the calculations involve a function, such as sum or product, the operands will be represented as their own `SolrJSONDump` objects in the `Details` slice.

It should be noted that the `Description` is not intended to be easily machine-readable. As a result, we have to do some potentially annoying parsing of its contents.

A `Description` that contains the substring `Failed to meet condition` indicates a non-match. (This appears to be a redundant signal to the `Match` field.)

A `Description` that contains the substring `weight(` indicates a component of the ranking algorithm that is a function of one or more constants, which relate to the result's contents, and a `boost` value, which is part of the ranking parameters. For example, the constants might represent how often the search query appears in the title, and the corresponding `boost` would represent how much influence the title has on the ranking. It is important to note that, given a `SolrJSONDump` of a `weight` that was calculated with a particular `boost`, we can simulate arbitrary values of

the `boost` by substituting the new `boost` and then mimicking the function that combined the constants and the `boost` into the `weight`.

`weight Description`s look like these examples:

```
1 weight(inboundlinks_anchortext_txt:botball in 2088) [
    DefaultSimilarity], result of:
2 weight(title:"hillary clinton cnbc" in 215) [ClassicSimilarity],
    result of:
3 weight(urlLength), product of:
```

There are two relevant things to parse out of a `weight`: the *field* and the *value*. The field is mandatory and represents the name of a variable being used as input to the ranking algorithm; in these examples, the fields are `inboundlinks_anchortext_txt`, `title`, and `urlLength`. The value is optional and represents the data associated with that variable, if applicable. For example, the second `weight Description` shown above indicates that the `weight` measures the ranking bonus that the result received for having the phrase `hillary clinton cnbc` in its title.

Not all `weight`s have a value, and not all values are interesting to us. This is because there are three main types of `weight`s that can appear:

1. Field Boost. This means that the search query appeared in a text field of the result, such as the title, and measures exactly how prominent the search query was in that text. The value is the token of the query that is being matched, and is usually a single word, unless double-quotes are used in the query. The value has no relation to YaCy's ranking rules, and can be discarded.
2. Boost Query. This means that YaCy's ranking rules have applied a bonus to a result because a discrete field had a predefined value. For example, this might be used to give a bonus to results whose `url_protocol_s` field has the value

`https`, which has the effect of rewarding websites for using encryption. The value is a critical part of YaCy's ranking rules.

3. Numeric Boost. This means that YaCy's ranking rules have applied a bonus to a result based on the value of a numeric field. No value is given in the `Description`.

String parsing easily shows which of these three situations applies. In practice, Field Boost and Numeric Boost situations can be treated identically once the field has been extracted.

Once a `weight` has had its `Description` parsed, its `Details` need to be parsed as operands to a function. The function is easily identifiable by the suffix of the `weight`'s `Description`. Common examples include `result of:` (identity), `sum of:` (sum), and `product of:` (product). The `Details` will be in one of three forms:

1. Another function; parsing can continue recursively.
2. A constant. This is any `SolrJSONDump` that doesn't have any `Details`, except for the third case (see below). Its `Value` needs to be remembered.
3. A boost. This has no `Details`, and has a `Description` of `boost`. Its `Value` is not remembered; instead, its `Value` will be replaced during simulation based on the ranking parameters being simulated.

3.2.2 `struct` Conversion

A `SolrJSONDump` is not in an ideal format for use in simulations. In particular, the string processing involved is unnecessarily CPU-intensive. We instead parse all `SolrJSONDump`s once, before any simulations occur, and convert them into a custom `struct` that we created for the purpose of being efficient to process. The protobuf3 (Google, 2018c) representation of this `struct` is:

```
1 message YaCyDump {
```

```
2  double value = 1; // Value that was calculated by Solr at dump time
       .
3
4  enum Op {
5    NO_MATCH = 0; // Results that don't match the search query at all
         .
6    CONSTANT = 1; // Ignore details, use value as-is.
7    BOOST    = 2; // This is a boost value that we may need to
         replace based on ranking parameters being simulated.
8    SUM      = 3; // Sum of details.
9    PRODUCT  = 4; // Product of details.
10   IDENTITY = 5; // Equal to first element of details.
11   POWER    = 6; // Equal of power of first 2 elements of details.
12   MAX      = 7; // Max of details.  Not differentiable.
13   FLOOR    = 8; // Floor of first element of details.  Not
         differentiable.
14 }
15
16 Op op = 2;
17
18 string boostfield = 3;
19 bool boostanyvalue = 4;
20 string boostvalue = 5;
21
22 repeated YaCyDump details = 6;
23
24 int64 dumptime = 7; // Unix timestamp when the dump was retrieved.
```

```
25
26   string urlandquery = 8; // URL + " " + query; used for lookup
        tables
27 }
```

The important feature of `YaCyDump` is that the `Description` has been broken up into `op`, `boostfield`, `boostanyvalue`, and `boostvalue`. `boostanyvalue` is true in the case of Field Boosts and Numeric Boosts; it's false for Boost Queries (when a `boostvalue` is present).

3.2.3 Constant Folding

To further minimize run-time of simulations, we apply constant folding (Wikipedia contributors, 2017b) to `YaCyDump`s. In particular, we apply the following folding operations:

- An identity function is replaced by its child.
- If a product or max function has more than one constant child, those children are folded into a single child.
- A product or max function with only one child is treated as an identity function.

Empirically, sum functions produced by YaCy do not benefit from the latter two folding operations.

3.3 Ranking Simulation

Given a ranking transparency dump and a set of ranking parameters, we implemented Go code that simulates the ranking algorithm using the specified parameters. This works by simply performing the math operation specified by the `Op`

attribute of a `YaCyDump`, except in the case that `Op` is `BOOST`, which indicates that the value is replaced by a value supplied by the ranking parameters.

3.4 Ranking Features Used For Learning

YaCy can utilize a variety of ranking components. Some examples:

- `text_t`: Ranks higher if the query appears in the text of the page.
- `title`: Ranks higher if the query appears in the title of the page. Useful since the title of the page may be more indicative of the overall topic, and less prone to spam/noise, than the text of the page.
- `synonyms_sxt`: Ranks higher if synonyms of the query appear in the text of the page. Useful since it will give results with similar meaning to the query, without making the user try multiple synonyms in the query.
- `crawldepth_i`: Ranks higher for low crawl depths. Crawl depth is the number of clicks needed to get to the page from the point where the crawl was started. Useful since YaCy users often start crawls at pages that they find interesting, which means that interesting pages tend to have lower crawl depths.

The full list of ranking components that were subject to learning in our testing is in Appendix 4.

3.5 Dataset Collection (Artificial/Testing Approach)

For the purpose of testing our learning algorithms, a set of 90 search queries were selected from the AOL Research dataset (Pass et al., 2006); these were divided into three subsets: training, validation, and test sets, with 31, 30, and 29 search queries, respectively. The queries from the AOL dataset were filtered in a few ways:

- Reject queries that indicate the user tried to type a URL. (Yes, they're AOL users, *of course* they regularly try to search for URL's.) Specifically:
  - Queries that end with `com`
  - Queries that end with `org`
  - Queries that end with `net`
  - Queries that begin with `www`
  - Queries that begin with `http`
  - Queries that contain a period, do not contain a space, do not start with or end with a period, and do not contain two adjacent periods
- Reject queries that consist of only a hyphen. (This is a common query in the AOL Research dataset for unknown reasons.)
- Replace `20` with a space. (A significant number of queries in the AOL Research dataset seem to have been escaped in this way for unknown reasons.) We don't perform this replacement when the `20` is adjacent to characters that are not letters or tabs, because queries like `battleships us 2005` appear in the dataset legitimately.

We experimented with manually filtering out misspelled queries, but the human effort involved was too much. We decided that the harm derived from learning against misspelled queries was likely to be less than the benefit of the increased sample size derived from no manual filtering.

We did **not** remove queries based on creepiness, offensiveness, or questionable legality. As a result, some of the queries we utilized were creepy, offensive, or of questionable legality. In the real world, a search engine's job is to return relevant results even in such cases; it therefore makes sense for our testing to reflect the real world.

These queries were fed to Startpage and DuckDuckGo, and the ranked top 100

results were transformed into a DAG (directed acyclic graph), in which an edge indicates that one result is more relevant than another.

3.6 Dataset Collection (Intended Real-World Approach)

We intend for a ranking DAG to be constructed based on real-world user data. We developed a WebExtension (Mozilla MDN contributors, 2018) that collects DAG data by implementing an upvote/downvote UI in search results pages. Detection of each result is based on a jQuery selector, and is therefore quite flexible. As a result, it can be easily adapted to many search engines; we coded initial support for YaCy and DuckDuckGo results pages. When a user clicks the upvote or downvote button, the search result and its adjacent result swap places with a smooth animation, and the WebExtension then notifies a Go application via the WebExtensions Native Messaging API (Mozilla MDN contributors, 2017). The Go application then saves the swap operation as a DAG edge, using the same Protobuf format as the artificial/testing approach.

It should be feasible to adapt the WebExtension to collect DAG data via other mechanisms as well, such as:

- UI in search results page to mark a result as "Exactly what I wanted", "Somewhat relevant", or "Completely irrelevant".
- UI after navigating to a result, to mark that result as "Exactly what I wanted", "Somewhat relevant", or "Completely irrelevant".
- Clickthrough data. Thorsten Joachims, Laura Granka, Bing Pan, Helene Hembrooke, and Geri Gay identify three heuristics (2005) for generating DAG edges from clickthrough data that are empirically accurate. In decreasing order of accuracy, these are:
  - The last-clicked result is more relevant than any non-clicked result that

appears anywhere above it in the search results page. (Strategy 2 in cited paper.)

  - Any clicked result is more relevant than any non-clicked result that appears anywhere above it in the search results page. (Strategy 1 in cited paper.)
  - Any clicked result is more relevant than a non-clicked result that appears immediately after it in the search results page. (Strategy 5 in cited paper.)

- User behavior heuristics, such as the DwellRank algorithm proposed by Blippex (2013a).

We intend to release the WebExtension as free software. We explicitly do **not** intend to distribute it on the official Chrome or Firefox extension stores, because those stores implement TiVoization (Free Software Foundation, 2017a), which we find highly unethical. Examples of distribution methods that we do plan to pursue:

- Source code distributed via a public Git repository.
- Bundling with official YaCy binaries; this would require coordination with YaCy.
- Packaging in official GNU/Linux distributions; this would require coordination with those distributions, and probably would require YaCy to be packaged as well.

These distribution policies also apply to the other WebExtension-related work described in this thesis.

3.7 Dataset Mitigation of Sybil Attacks

In a search engine which is trained by its users, there is an incentive for bad actors to try to gain undue influence over the training process. Examples of such bad actors

include:

- Spammers trying to boost their own websites' ranking.
- Other search engines trying to sabotage their competition.
- Corporate or government censors trying to bury targeted websites.

These types of bad actors might try to perform a Sybil attack (Wikipedia contributors, 2018g; Douceur, 2002). There are several pre-existing methods intended to combat Sybil attacks, including centralized identity, proof of work, and web of trust. Below we will examine these methods.

3.7.1 Centralized Anti-Sybil

Centralized anti-Sybil systems utilize a trusted third party to keep track of which people are using the system, and make sure that each user is weighted equally. Centralized systems might be in the form of verifying a real-world identity such as a government-issued ID, but identities need not be in the form of a name on a birth certificate. IPv4 addresses are often used as an approximation of real-world identity, because the effort involved in a single person obtaining multiple IPv4 addresses is sufficiently large, and the typical number of people who share a single IPv4 address is sufficiently small, that anti-Sybil algorithms can frequently treat IPv4 addresses as people. Phone numbers often play a similar role to IPv4 addresses. CAPTCHAs (Wikipedia contributors, 2018c; von Ahn et al., 2003) may also be used to try to raise the cost of Sybil attacks. Another common approximation is to use email addresses from providers who are known to utilize some kind of anti-Sybil system of their own.

This approach has two serious issues. First, these systems generally are not anonymous (they are sometimes pseudonymous, but even pseudonymity is unacceptable from a privacy standpoint). While CAPTCHAs can theoretically be anony-

mous, they are highly annoying to most legitimate users, they are easily outsourced to vendors of cheap human labor, and AI is inching closer to matching human performance. As a result, when CAPTCHAs are used, they are usually in combination with some other system, in a way that penalizes users who value their privacy; Cloudflare's discrimination against Tor users (Perry, 2016) is an example of this. Second, the reliability of the Sybil prevention is entirely dependent on the trusted third party; the third party could, if they wished, perform the same kinds of manipulation of ranking that we are trying to prevent.

As a result of these issues, we reject centralized anti-Sybil mechanisms.

3.7.2 Proof of Work

Instead of relying on a trusted third party to act as a rate limiter, there are decentralized rate-limiting methods that can be independently and cheaply verified. The classic method is proof of work (PoW), the most well-known implementation of which is Hashcash (Wikipedia contributors, 2018f; Back, 1997; Back, 2002). To generate a Hashcash proof that work was expended to create a message, extra data must be attached to the message such that the hash of the message and extra data is smaller than a given target; this target determines the difficulty of the Hashcash challenge. Generating a Hashcash proof is computationally expensive, as it requires partially preimaging a hash function. However, to verify a Hashcash proof, only a single hash function evaluation is needed, making the verification process extremely cheap. Hashcash is probably best known as the basis of Bitcoin (Nakamoto, 2009) mining, although it was originally proposed as an anti-spam tool. Various other systems similar to PoW have been proposed. For example, proof of stake (PoS) uses ownership of a cryptocurrency as a rate-limiting method.

PoW is a particularly powerful tool, because in the context of a Nakamoto blockchain, it forms a decentralized solution to the Byzantine Generals' Problem

(Wikipedia contributors, 2018b). This is why PoW is widely used in decentralized systems that must form a global consensus, such as currencies like Bitcoin or naming systems like Namecoin (Durham et al., 2018). (Note that PoS, by itself, cannot be used as PoW can to form a decentralized consensus system, but in the presence of a PoW-based consensus system, PoS is not without its use cases.)

As powerful as PoW is, it has some significant drawbacks. PoW tends to be more efficient at large scale, which has resulted in centralization of Bitcoin mining operations. PoW also rewards users who can afford more computing power, who in practice may very well be the corporate and government actors whom we are trying to avoid giving undue influence.

Given these drawbacks, it's worth noting that a search engine's training system does not need to be Byzantine fault tolerant. If different users have different views of the training data, that doesn't invalidate the system's usefulness like it would for systems like Bitcoin and Namecoin. Since PoW's primary advantage is not particularly useful to us, and it has issues of its own, we chose not to utilize PoW for our Sybil prevention.

3.7.3 Web of Trust

Web of Trust (WoT) (Wikipedia contributors, 2018h; Zimmermann, 1994) was originally proposed by Phil Zimmermann in 1992. A WoT allows each user to choose trusted friends, who can choose their own trusted friends, etc. If a trusted path is formed between two individuals, then they have a positive trust value. This differs from Byzantine fault tolerance in the sense that users Alice and Bob may have different trust values for the user Carol. This is useful for a search engine, if we assume that users will choose friends with similar interests (social searching). Social graphs are usually highly resistant to Sybil attacks, since users will usually not friend large numbers of sockpuppets.

WoT can be utilized for ranking algorithm evaluation via the following algorithm:

1. Alice sends a ranking algorithm to her friends.
2. Alice's friends each send the algorithm to their friends. (Repeat this step depending on how many degrees of separation are desired; higher degrees of separation will scale more poorly but will have a higher sample size.)
3. Alice's friends' friends each evaluate the ranking algorithm against their own training data, and return an error value to Alice's friends.
4. Alice's friends each evaluate the ranking algorithm against their own training data, combine it with the error values they received from their friends (via an outlier-resistant measure of central tendency, giving their own data higher weight than each of their friends' data), and return an error value to Alice.
5. Alice evaluates the ranking algorithm against her own training data, combines it with the error values she received from her friends (via an outlier-resistant measure of central tendency, giving her own data higher weight than each of her friends' data), and now has a resulting error value.

This WoT construction has two interesting properties:

1. Raw ranking transparency dumps are not shared, even between friends. As a result, the only data that a user will get regarding their friends' ranking transparency dumps is derived from aggregating all of that friend's transparency dumps. We believe (but have not proven) that, assuming a reasonably large set of ranking transparency dumps per user, it should be impractical for Mallory to deduce Alice's search queries or browsing history via this WoT design.
2. Users only communicate with their friends, and do not ever learn the identities of their friends' friends—not even pseudonyms or number of friends. This is extremely useful for privacy, since we believe that social graph metadata should be considered private information.

Mike Perry has written a critique of WoT for the authentication of OpenPGP public keys (2013); our WoT construction avoids all three of Mike's concerns:

1. Social graph metadata and identity information are not leaked by our WoT construction.
2. Our WoT construction uses an *undirected* social graph, which significantly reduces the risk of a "strong set" of identities emerging. This avoids the CA-like problem that Mike identifies.
3. Each user's data (both their ranking transparency dumps and their social graph) stays in local storage; it is not distributed to the entire set of users and therefore does not run into the storage-related scalability problems that Mike identifies.

Analysis of Friend Weighting

It should be noted that the weighting in this algorithm is inherently approximate, because Alice does not take into account the DAG edge sample size of each of her friends, nor the number of friends that each friend has. If we were to assume that all users in Alice's social graph are non-malicious (more specifically, that all users in her social graph are not falsifying their DAG data or their social graph), then Alice would definitely want to weight her friends' answers by total number of users who provided the data for that friend (or, even better, by total number of search queries or DAG edges that were stored by all of those users). However, our threat model expects that a small number of users will be falsifying their data (with decreasing probability of falsification as degrees of separation from Alice increases), and we therefore do not perform such weighting.

The effect that this has on the system is that some DAG data will be disproportionately represented in the overall learning vector. This might decrease the optimality of the eventual converged learning result (though we conjecture that this

will not be a very harmful effect), and it might also decrease the speed at which the learning converges (we conjecture that the effect here is more likely to be strong, but we also believe that the slower convergence time is a reasonable price to pay for the improved security).

One plausible approach to reducing the impact of users with an extremely low number of friends, or an extremely low number of local DAG edges, is to allow users to reply with an "I'm still bootstrapping, so please exclude me from this calculation" error when a friend asks them to perform an evaluation. This does leak one bit of information to a user's friends about their number of friends, as well as one bit of information about their number of DAG edges, but we consider it unlikely that those two bits of information will have any significant impact on privacy.

Social Bootstrapping

Since it may take some time to learn high-quality ranking parameters from a random starting point, we suggest that users may wish to share their "hall of fame" learning results (the ranking rules they've learned so far that performed the best) with their immediate friends. As a result, new users can use their friends' ranking parameters as a starting point. Usage of the social graph for this purpose eliminates the risk of Sybil attacks for default ranking rule choice, and also takes advantage of the intuitive conjecture that Alice's ideal ranking parameters may be closer to those of her friends than to the general population (this conjecture is based on the logic that social graph proximity tends to correlate with common interests, and a search engine's goal is to return information that matches a user's interests).

Social Graph UI

It should be noted that YaCy does not currently have any UI for adding friends; such a UI would need to be implemented, either in YaCy itself or in a helper application,

in order for WoT to be used with YaCy.

3.8 Evaluation of Dataset Size

Adam Pash published an article at Lifehacker (2011) explaining how Google Search users can check how many searches they've performed over time. Pash asked readers to give their data in the comments section. The full set of users' replies is in Appendix 5.

Of the subset of the above users who reported a total count:

$Mean(9500, 20688, 40000, 23932, 967 + 3200 + 4071 + 3634 + 2642 + 3172 + 985, 19822, 51000, 10876, 13948, 9806, 41049) = \frac{259292}{11} = 23572$

According to Pew Research in 2014 (Smith, 2014), the mean number of Facebook friends an adult user has is 338 (median 200).

It then follows that if 1 degree of social graph separation is used, we get:

23572 mean total searches (via Lifehacker dataset) $\cdot$ (1 self user + 338 Facebook friends) $= 7,990,908$ total searches within 1 degree of separation in social graph.

And, for 2 degrees of separation, under the assumption that the social graph is a tree rooted at Alice (which overestimates the user count):

23572 mean total searches (via Lifehacker dataset) $\cdot$ (1 self user + 338 Facebook friends)^2 $= 2,708,917,812$ total searches within 2 degrees of separation in social graph.

For comparison, the artificial Startpage-derived dataset that we tested with contained 90 total searches. We conjecture that 1 degree of separation should be sufficient for learning purposes, and perhaps even 0 degrees of separation. This greatly simplifies the WoT requirements, since it eliminates the need to hide users' social graph (and, in the case of 0 degrees of separation, eliminates the need for WoT entirely). Validating these conjectures based on theoretical statistics would be un-

reliable, because there is likely to be a correlative effect among searches that a single user performs, and also a correlative effect among searches that a user and their friends perform. Without knowing what correlation exists, it is not feasible to determine whether the resulting ranking rules will be useful for the general population. Since the Social Bootstrapping feature that we proposed requires that Alice's learned ranking parameters be useful for her friends, who might then share them with their friends recursively, it is highly beneficial if Alice's learned ranking rules are at least somewhat useful for the general population.

3.8.1 Bytes Stored and Transferred

As a rough estimate of bytes stored, the "Artificial/Testing Approach" dataset, which consists of 90 searches, with the top 100 Startpage results all ordered from most to least relevant, uses 36.8 MB of storage in Protobuf format. This is likely to be a strong overestimate (in terms of bytes per search query) compared to the "Intended Real-World Approach" since most search queries performed by actual users will produce only a few DAG edges.

In terms of bytes transferred between friends, each request consists of a set of YaCy ranking parameters, which uses 3.0 kB when serialized as JSON. Each response consists of an error value (a 64-bit floating-point number) and a learning vector (same size as the YaCy ranking parameters in the request).

3.9 Learning with Ranking Transparency

It is our intention that our data collection and analysis tools be considered to be a "ranking laboratory" where anyone can apply their own ranking algorithm ideas, not merely a "ranking machine learning algorithm" that everyone is forced to use. Machine learning is a diverse and rapidly evolving field, and we do not make any

claims as to the "best" machine learning approach for producing high-quality ranking. However, as an informational example to verify that our dataset was usable, we implemented learning from ranking transparency dumps based on the principle of gradient descent. A DAG was constructed, where each node is a ranking transparency dump for a given query and URL, and each edge represents the desired ordering of two URL's for a given query. For each edge in the DAG, the following algorithm was used:

1. Simulate each node in the edge using the current ranking rules.
2. If the nodes' simulated ranking is ordered correctly, do not learn for this edge; terminate algorithm and move onto the next edge.
3. Calculate difference in simulated ranking between the two nodes; this is the error for the edge.
4. Calculate partial derivative of error with respect to each ranking parameter; this is the learning vector.
5. Multiply learning vector by learning rate, and increment the ranking parameters by the result.

It should be noted that technically, YaCy's ranking algorithms are not differentiable, because they contain functions such as `floor` and `max`. We approximated a partial derivative by incrementing each ranking parameter by a small delta, and measuring the resulting change in the error.

The error function is simply the fraction of DAG edges that are ordered correctly, although the range [0, 1] is internally represented as [1, -1], so -1 indicates that the ranking rules produce identical results to the training data, 1 indicates that the ranking rules produce the opposite ordering, and 0 indicates ranking rules that are equivalent to random guessing. It should be noted that the error will occasionally get worse even though learning is occurring in the correct direction; this happens

when a "very wrong" edge is being corrected at the cost of causing a "slightly right" edge to turn into a "slightly wrong" edge. Generally, the resulting "slightly wrong" edge will correct itself later in learning.

This learning algorithm was tested using DAG data derived from the "artificial/testing approach" of scraping Startpage. In reality, the "real-world approach" of using user-generated data would be used, but obtaining sufficient user-generated DAG data for our experiments would have been a significant undertaking. Since both data collection approaches save their data into the same Protobuf-based format, a learning algorithm designed for Startpage-generated DAGs should work fine with user-generated DAGs without significant changes.

We consider reporting the quality of learning results to be out of scope of this work, since the gradient descent experiment is not proposed to be a high-quality learning algorithm/implementation; it is simply a mechanism to verify that our data collection and analysis tools are usable. However, we do note that our gradient descent code does produce a learning vector that results in an improved error value over time.

Chapter 4

Other Relevance Improvements

In this chapter, we describe our investigations into two different YaCy issues (one design flaw and one implementation bug) that severely damaged ranking quality, and the fixes for both issues that we submitted to the YaCy developers. We then discuss several mechanisms for expanding YaCy's distributed index to cover relevant URL's. One such mechanism is to start crawls from all URL's visited in the user's web browser; we explain why even though YaCy nominally supported this already, the approach taken by YaCy is inferior to our method. Another such mechanism is to start crawls on the results of mainstream search engines whenever a search in YaCy, or another search engine, is initiated; like the previous mechanism, we explain why YaCy's nominally supported method for achieving this does not work reliably. A third mechanism is to crawl results for searched queries that have previously been deliberately targeted for censorship, via the Lumen database. Finally, we introduce a new ranking criterion, which uses Wikipedia's external links as a curation mechanism for identifying high-quality websites.

4.1 Investigation of YaCy's Result Sorting

While implementing ranking transparency into YaCy, an interesting issue became apparent: the JSON search interface wasn't sorting the results by ranking. This falls into the "very bad" category of bugs, for fairly self-evident reasons. We later discovered that YaCy user Davide had independently discovered this issue (2015). However, it was unclear whether this only affected the JSON interface or whether

it also affected the HTML interface, which would make it substantially worse. We therefore implemented the capability to display ranking data in the HTML interface; this was submitted to the YaCy developers and merged by YaCy developer luccioman (Rand, 2016e).

Unfortunately, this demonstrated that the HTML search interface was also failing to sort results by ranking. Paraphrasing a user on Freenode's #yacy channel, "Have you ever thought that YaCy's ranking was so bad that it must be literally random? Turns out... you were right!"

We did some code review to identify the cause of the sorting failure; we determined that the cause was that YaCy sends the results to the user immediately after receiving them from remote peers, which means that the primary determining factor in result sorting is the network latency to the peer that returned the result. The full analysis that we submitted (Rand, 2016f) to the YaCy developers is in Appendix 1.

4.1.1 Fixing the bug

We implemented a fix based on suggestion #2 (JavaScript-based sorting) as described in Appendix 1. JavaScript-based sorting in the user's web browser resolves the issue, since new results from remote peers can be inserted into any position in the results list, regardless of when a peer returned a given result. We submitted the fix as a PR to the YaCy developers (Rand, 2017a); it was merged by YaCy developer luccioman.

4.2 Investigation of YaCy's Usage of Boost Queries

YaCy uses a Solr feature called Boost Queries, which boosts scores of results if they fit given criteria. For example, the Boost Query `crawldepth_i:0^0.8` boosts by 0.8 all results for which the `crawldepth_i` attribute is equal to `0`. We noticed a bug in

YaCy's handling of Boost Queries that caused incorrect behavior when more than one Boost Query is active. Given that YaCy's default settings include two Boost Queries, this is particularly unfortunate. We debugged the issue and determined that the cause was an incorrect encoding of the separator between multiple Boost Queries when YaCy is communicating with Solr. The full analysis that we submitted (Rand, 2016c) to the YaCy developers is in Appendix 2.

4.2.1 Fixing the bug

We submitted a PR to fix the Boost Query issue (Rand, 2016b). The PR was reviewed by YaCy developer reger24; after some discussion between reger24 and us about an implementation detail, reger24 merged the PR, fixing the bug.

4.3 Growing YaCy's Index

YaCy's global index is likely to be considerably smaller than those of centralized search engines, since centralized search engines expend significant resources to crawl large amounts of the Web. It would be beneficial to have automated methods of filling YaCy's index.

4.3.1 Visited Web Pages

YaCy contains a built-in HTTP proxy that indexes pages that pass through it. This is intended as a method of filling YaCy's index with pages that YaCy's users actually care about. However, an HTTP proxy has a few drawbacks as a method of accomplishing this goal:

- It cannot index sites that use TLS, since the proxy isn't aware of the TLS-protected content. (Technically it would be possible for the proxy to intercept the TLS connection, but this is a bad idea for security reasons.)

- It chooses not to index any pages that have cookies or query strings, because those pages might contain private data.
- It can only index with depth 0, since it doesn't actually crawl any links on the page.

We implemented an alternative approach using a Greasemonkey script (Rand, 2015). On each page load, the script reads the current `window.location.href`, and submits it to YaCy's API as a crawl. This has the following advantages:

- TLS "just works", since YaCy makes its own TLS connections rather than trying to read the browser's TLS connections.
- Cookies aren't a privacy problem, since YaCy doesn't have access to the browser's cookies.
- Query strings can be optionally stripped out by the Greasemonkey script, so if the query string isn't essential to the content, the content can be indexed without privacy problems.
- The user can choose the crawl depth. (We chose a default of 1.)

More recently, we rewrote the Greasemonkey script as a WebExtension (combining it with the WebExtension that collects ranking data).

4.3.2 OpenSearch Heuristics and RSS-Bridge

YaCy supports a feature called OpenSearch Heuristics, which hooks YaCy search requests, retrieves an OpenSearch (Wikipedia contributors, 2017d) RSS feed from a user-defined source, indexes the results, and then includes the results in the YaCy results (mixed in with the local and P2P results, using the same ranking algorithm as if a YaCy peer had returned the results). This is theoretically useful as a way to grow YaCy's index using third-party search engines. However, most centralized search engines do not offer OpenSearch RSS feeds. This is unsurprising, since offering

OpenSearch RSS feeds would reduce centralized search engines' ability to leverage their monopoly.

Luckily, RSS-Bridge (RSS-Bridge contributors, 2018) is a project which facilitates generating RSS and JSON feeds for websites which don't offer such feeds. RSS-Bridge has a significant number of bridges available, and creating new bridges is relatively straightforward.

4.3.3 DuckDuckGo and Startpage Bridges

RSS-Bridge already had bridges for DuckDuckGo and Google, which would seem to be beneficial for YaCy. However, both the DuckDuckGo and Google bridges only returned results sorted by date (most recent first); they did not support sorting by relevance. This is problematic for use cases where a search is not frequently performed on the YaCy network, or where the YaCy P2P index does not contain very many results for a search. We modified the DuckDuckGo bridge to support sorting by relevance; this has been submitted as a PR and merged into RSS-Bridge (Rand, 2016a). We also implemented a Startpage bridge, which is useful since Startpage tends to have better privacy policies than Google, while providing similar results.

4.3.4 YaCy's OpenSearch Heuristics Are Broken

While we were inspecting YaCy's index, we made a surprising discovery. YaCy's OpenSearch Heuristics actually generate index data that *only* contains the data from the RSS feed—the URL's listed in the RSS entries aren't actually crawled. This is probably extremely harmful to any effort to get an accurate ranking, and it is likely that results that were indexed via an OpenSearch Heuristic will exhibit systematic bias in ranking behavior.

As a workaround, we added this functionality directly to our WebExtension.

When the WebExtension detects that a search result page has been loaded, it retrieves the search results for other supported search engines via AJAX, and submits the resulting URL's to YaCy's crawler. This was relatively straightforward, since RSS-Bridge uses CSS selectors to parse the data, which is also easy to perform in JavaScript, in which WebExtensions are written.

4.3.5 Lumen Bridge

Using DuckDuckGo and Startpage with RSS-Bridge to bootstrap YaCy's index seems like a good idea, but there is an issue: centralized search engines often censor results. This means that if YaCy's index is bootstrapped via centralized search engines, there will be an inherent bias in YaCy's index against search results that have been subject to censorship by centralized search engines.

Our solution is to implement an RSS-Bridge for Lumen (Seltzer, 2018) (formerly known as Chilling Effects). Lumen is a database of censorship requests to search engines and other services. Using the Lumen RSS-Bridge with YaCy attempts to ensure that any attempt to censor search results from centralized search engines will increase the chance that those results will be indexed by YaCy; this is an example of the Streisand Effect (Masnick, 2005).

4.4 Social Ranking with Wikipedia Data

YaCy developer Michael Christen suggested on the YaCy forum using popularity metrics for websites provided by Twitter via the YaCy spinoff Loklak (Christen, 2016). We noticed that Wikipedia is another source for such data. All Wikimedia content is libre, and Wikimedia provides separate database dumps of each table, updated twice per month (Wikimedia Foundation, 2018); these are licensed under the GNU Free Documentation License and the Creative Commons Attribution-Share-

Alike 3.0 License (Wikimedia Foundation, 2017). The `externallinks` table is the one we want; the gzip-compressed English Wikipedia `externallinks` table from September 1, 2016 is only 2.3 GB – well within the ability for most YaCy users to download twice per month.

One question was how to integrate external ranking with YaCy. We found that Solr already has a built-in mechanism for this: the `ExternalFileField` data type (The Apache Software Foundation, 2016).

The other question was how to generate a key/value pair file from the Wikipedia database dump. We tried several methods. Initially, we tried importing the Wikipedia dump directly into MySQL, with the intent to use MySQL queries to obtain frequency data. However, MySQL was sufficiently heavy in resource usage that we abandoned this idea – the import process saturated I/O usage, making the computer unusable for other work, and the import process, which would need to be redone every month, was projected to take more than 24 hours. We then tried converting the MySQL dump into a SQLite dump using `mysql2sqlite` (dumblob, 2018). The conversion process was very quick when using `mawk`; however, importing the converted dump into sqlite was still problematically slow: it took many hours, and saturated I/O during this time. In addition, the sqlite database that was produced was 56.2 GB, which is potentially prohibitive.

At this point, we noticed something while looking at the MySQL database dump: the format of MySQL dumps is actually fairly similar to that of Python. In fact, replacing the leading `INSERT INTO ‘externallinks‘ VALUES` with `[` and replacing the trailing `;` with `]` in each of the MySQL lines that contains relevant data results in a valid Python literal (specifically, a list of tuples). StackOverflow user Niklas B. pointed to a Python function, `literal_eval`, which can parse such data (2012). StackOverflow user ShadowRanger mentioned an efficient way to count word frequencies in Python (2017), which was sufficient for us to implement a Python

script to convert the MySQL database dump into an `ExternalFileField` key/-value pair file. The `urllib.parse` library was used to convert full URL's into domain names, as per a hint by StackOverflow user Gerard (2017). This was substantially more efficient than using MySQL or SQLite: converting the `.sql` file to an `ExternalFileField` file, with sorting, took 1 hr, 23 min, 28 sec. It only used one CPU core, and did not use significant I/O or RAM, meaning that the system was completely usable for other work while this was running. The resulting file was only 90.3 MB.

We hooked up the `ExternalFileField` file to YaCy's embedded Solr instance, rebooted YaCy, and all Solr queries made against that embedded Solr instance were able to successfully use the Wikipedia external link count as a ranking boost. Qualitatively, we did not notice any slowdown while making queries. There may have been a slight slowdown in booting YaCy, but it was not problematic. Our code for this is publicly released (Rand, 2017c).

Chapter 5

Privacy Improvements

In this chapter, we introduce a new tool that uses `strace` to automatically identify proxy leaks in a way that is compatible with both manual testing and automated CI (continuous integration) testing. We then cover a proxy leak that it discovered in YaCy, and make recommendations for improving YaCy's proxy usage for maximal anonymity. We also perform a preliminary audit of the YaCy P2P protocol for application-layer privacy leaks, and make initial recommendations for improving the situation.

5.1 Proxy Leaks in YaCy

Privacy-conscious users typically route application traffic through a proxy such as Tor (The Tor Project, 2018b). If a subset of application traffic is sent without being routed through the configured proxy—such a situation is referred to as a *proxy leak*—this can be dangerous to privacy. Although it is possible to use `iptables` to transparently route all traffic from a machine through a configured proxy, as Whonix (Whonix contributors, 2018b) does by default, this will usually violate stream isolation (Whonix contributors, 2018a; Perry, 2011), and is not a substitute for properly configured proxy support in applications.

YaCy's UI has a configuration option to use an HTTP proxy. We were curious whether this feature suffered from any proxy leaks. Unfortunately, there weren't any readily available tools for easily detecting proxy leaks in an automated way. So, we had to create one.

5.1.1 Heteronculous: A Proxy Leak Detector That (Hopefully) Sucks Slightly Less

As a starting point, we noticed pabouk's answer on the Tor StackExchange (2013), which suggested that `strace` (strace contributors, 2018) could be used to observe socket-related syscalls made by an application, and thus determine whether they are communicating with any IP address other than the desired proxy.

We therefore created a Bash script that automates using `strace` for detecting proxy leaks. Some useful features that we included:

- Pipe `strace`'s output through `grep` to filter out any socket calls that don't indicate proxy leaks.
- Based on a configuration environment variable, optionally consider any usage of IP sockets to be a proxy leak. This is useful when the application being tested is intended to communicate with the proxy over a Unix domain socket, which is considered to be a best practice (Tor Wiki contributors, 2017).
- Use configuration environment variables to specify a whitelist of IP addresses and ports that the application is permitted to interact with. This whitelist would include the proxy, and sometimes also other services such as the Tor control port.
- Automatically follow forked processes, via the `-f` option to `strace`. This allows us to `strace` the YaCy startup Bash script and get useful results for the JVM that the Bash script launches.
- Interleave `grep`ped `strace` output with the output of the target process in real-time. This required using `stdbuf` to disable I/O buffering in `grep`.
- Exit code of the Bash script indicates an error (1) if and only if `strace` detected proxy leaks. This facilitates usage in automated test suites, such as for CI (continuous integration) purposes.

We named this bash script *Heteronculous* (etymological notes are in Appendix 3). Heteronculous has been successfully used in the wild to fix a DNS proxy leak in the XMPP client Gajim (Horist, 2017). We also used Heteronculous to successfully reproduce a proxy leak in the PySocks library (Anorov, 2015), as well as confirming that the proposed fix for that proxy leak (Fitblip, 2012) worked as intended.

5.1.2 Testing YaCy for Proxy Leaks

In our initial testing, Heteronculous quickly noticed that remote Solr queries leaked outside of the configured HTTP proxy. However, Heteronculous also confirmed that configuring the Java VM to tunnel over SOCKS instead of using YaCy's HTTP proxy support fixed the proxy leaks; this configuration was suggested to us by Linker Lin (2016).

We intend to perform more thorough proxy leak testing of YaCy with Heteronculous in the future, as we move toward built-in Tor/SOCKS support (see next section).

5.2 Tor and SOCKS support in YaCy

Although the current approach for SOCKS support, in which the Java VM is configured to tunnel over SOCKS, works reasonably well, it is definitely not ideal, for two reasons:

- It prevents YaCy from doing any application-specific SOCKS configuration, such as username/password-based SOCKS authentication for stream isolation (Perry, 2011).
- It doesn't handle the Tor control port, which is needed for configuring a Tor onion service (The Tor Project, 2018c; The Tor Project, 2018a).

In the course of unrelated work in the blockchain development space, we happened to learn of the NetLayer library by Bernd Prunster (2018), and we believed that it was suitable enough for this use case that we did some security review of it—in particular, checking the stream isolation implementation (Rand, 2017b). We were very pleased with Bernd's response to our security review, and we currently believe that the best approach for YaCy Tor support is to integrate NetLayer into YaCy with the following functionality:

- All outgoing traffic should be routed over Tor by default.
  - Outgoing P2P traffic should be independently configurable from outgoing crawl traffic, because only P2P traffic reveals your searches to the P2P network, and some websites block crawl traffic that comes from a Tor exit relay.
- All incoming traffic should be routed over Tor onion services by default.
- Each search query should be stream-isolated from other queries, and from non-query traffic, via SOCKS authentication.
- Each peer should be stream-isolated from other peers. This is actually the default for onion services, so it's not actually necessary to do via SOCKS authentication unless we want to continue supporting non-onion-service peers. For non-onion-service peers, this policy is important to mitigate the risk of malicious Tor exit relays.

This is, of course, subject to peer review by the anonymity network community.

5.3 Application-Layer Privacy Leaks in YaCy

We performed a non-extensive audit of application-layer privacy leaks in YaCy based on the YaCy protocol documentation (YaCy Wiki contributors, 2016). It should

be noted that most of these attacks are not particularly relevant until Tor support, with onion service and stream isolation support, is added, since deanonymizing non-Tor users is easy without needing these attacks. For this reason, we have neither tested these attacks against real-world YaCy nodes nor attempted to implement fixes for them (we think Tor support is higher-priority); we are simply going off of the protocol documentation.

1. Connect to all YaCy nodes, get them to submit searches to me. (`search.html`)
    a. `myseed` may be fingerprintable
    b. `query`, `exclude`, `urls`, `abstracts` will yield hashes of words/URL's of interest, such as `tor` or `http://www.wikileaks.org/`
    c. `prefer`, `filter` may have keywords of interest in their regexps
    d. `count`, `time`, `maxdist` may be fingerprintable
    e. source IP address may be fingerprintable if not an anonymizing proxy
    f. source IP address may be fingerprintable if it is an anonymizing proxy but the circuit has not been rebuilt
2. Send search requests to all YaCy nodes. (search.html)
    a. `version`, `uptime` may be fingerprintable
    b. destination IP address may be fingerprintable
    c. destination `.onion` address may be fingerprintable
    d. request `urls` or `filter` can query for URL's of interest
    e. request `query` can query for keywords of interest
    f. response resource metadata may be fingerprintable (`load` in past 24 hours may also indicate that the node was responsible for crawling the resource)
3. `hello` leaks
    a. request `seed`, `count` may be fingerprintable

b. response `version`, `uptime`, `mytime`, `seed[0]` may be fingerprintable

c. source IP address may be fingerprintable

d. destination IP/`.onion` address may be fingerprintable

4. Connect to all YaCy nodes, get them to submit searches to me. (`solr/select`)

   a. The query will yield words of interest, such as `tor`

   b. Various Solr parameters may be fingerprintable, such as the ranking parameters if they are changed from the default

   c. source IP address may be fingerprintable if not an anonymizing proxy

   d. source IP address may be fingerprintable if it is an anonymizing proxy but the circuit has not been rebuilt

   e. There may be other sources of Solr-based privacy leaks, since we didn't carefully examine this API

5. Send search requests to all YaCy nodes. (`solr/select`)

   a. destination IP address may be fingerprintable

   b. destination `.onion` address may be fingerprintable

   c. The query can look for keywords or URL's of interest.

   d. There may be other sources of Solr-based privacy leaks, since we didn't carefully examine this API

6. GSA equivalents of above Solr leaks

   a. We didn't examine the GSA API, but we expect that it's likely to have similar risks as the Solr API

7. Remote crawl URL's (`urls.xml`) may leak data and/or be fingerprintable.
8. It may be possible to deploy tracking cookies by inserting a specific index entry into a remote peer's Solr and/or DHT. Open questions about this attack:

a. How fast would such a tracking cookie propagate through the network (making itself useless)?
b. What if a tracking cookie consists of multiple index entries?
c. Would such a tracking cookie allow tracking even if the remote peer changes IP addresses, changes Tor exit relays, and/or changes `.onion` domains?

In general, we suspect that a large fraction of the above leaks can be fixed by the following changes to YaCy:

1. Prerequisite: move all connections to Tor onion services.
2. Run two local indexes: a private and public index.
3. The private index is filled by searching and crawling.
4. The public index is filled by DHT.
5. The private index does not accept incoming connections; it only shares its data via outgoing connections.
6. The private index adds an extra field for each index entry, which keeps track of stream isolation data (specifically, the first-party domain of the crawl start URL that resulted in the entry being indexed). The private index never shares two index entries that have different stream isolation data over the same stream.
7. The public index accepts incoming and outgoing connections.

This eliminates a substantial amount of attack surface involving fingerprinting of users by looking at the contents of their index. Again, we stress that **such a solution may not be sufficient**, and **it is not currently deployed**, because we consider Tor support to be higher priority.

Chapter 6

Conclusion

YaCy shows significant potential as a replacement for current search engines, with major potential advantages in the areas of censorship resistance, privacy, decentralization, transparency, and software freedom. We have demonstrated improvements in relevance and privacy that we believe bring YaCy closer to realizing this vision. Areas of future work include:

- Get ranking transparency merged to upstream YaCy.
- Improvements to the dataset-collection WebExtension, to collect a wider range of data.
- Experiments with different machine learning algorithms using the dataset collected by the WebExtension.
- Experiments to empirically determine whether a single user's dataset is sufficient for machine learning purposes, or whether a social graph degree of separation of 1, or 2, produces significant benefits.
- Implementation and deployment of the web-of-trust design.
- Implement mechanisms to verify authenticity of index data received from peers.
- Make YaCy's OpenSearch heuristics actually crawl the result pages, thereby allowing the WebExtension to remove that functionality.
- Further experiments with integrating Wikipedia-based ranking, such as determining how heavily it should be weighted.
- Implement the proposed Tor support using NetLayer, including stream isola-

tion.

- Conduct further proxy leak testing with Heteronculous, and also test Heteronculous against a wider range of software.
- Implement application-layer privacy improvements such as the proposed dual-index design.

References

Anorov, 2015. Monkeypatch socket DNS resolving functions. Available at: https://github.com/Anorov/PySocks/issues/22.

Back, A., 1997. [ANNOUNCE] hash cash postage implementation. Available at: http://www.hashcash.org/papers/announce.txt. Also available at: https://web.archive.org/web/20180425110500/http://www.hashcash.org/papers/announce.txt.

Back, A., 2002. Hashcash - a denial of service counter-measure. Available at: http://www.hashcash.org/papers/hashcash.pdf. Also available at: https://web.archive.org/web/20180502140708/http://www.hashcash.org/papers/hashcash.pdf.

Barlow, H., 1989. Unsupervised learning. *Neural Computation*, 1, pp.295–311. Available at: https://sci-hub.tw/10.1162/neco.1989.1.3.295.

Benazera, E., 2014a. Commits – beniz/seeks. Available at: https://github.com/beniz/seeks/commits/master.

Benazera, E., 2014b. Seeks. Available at: https://beniz.github.io/seeks/.

Benazera, E., 2016. Beniz comment on "Seeks nodes are not usable". Available at: https://github.com/beniz/seeks/issues/28#issuecomment-180980746.

Blippex, 2013a. A search engine built by the crowd, that does not suck. Available at: https://blippex.github.io/updates/2013/07/11/launch.html.

Blippex, 2013b. Blippex – blog. Available at: https://blippex.github.io/.

Camp, L., 2016. Google will steal this election & how - Dr. Robert Epstein interview. Available at: https://www.rt.com/shows/redacted-tonight-summary/339014-us-election-internet-politics/.

Caruana, R. & Niculescu-Mizil, A., 2006. An empirical comparison of supervised learning algorithms. *23rd international conference on Machine learning*, pp.161–168. Available at: https://sci-hub.tw/10.1145/1143844.1143865.

Christen, M., 2016. Social media ranking. Available at: https://github.com/yacy/yacy_forum_archive/blob/1d90ffa7ef5523303c1cea6f

1acfddeab129a187/content/2016-06-0718:02:03YaCyCoding%26Ar.md.

Christen, M., 2017. YaCy – the peer to peer search engine. Available at: https://yacy.net/en/.

Damon, A., 2017a. Evidence of Google blacklisting of left and progressive sites continues to mount. Available at: https://www.wsws.org/en/articles/2017/08/08/goog-a08.html.

Damon, A., 2017b. Google blocked every one of the WSWS's 45 top search terms. Available at: https://www.wsws.org/en/articles/2017/08/04/goog-a04.html.

Damon, A. & Niemuth, N., 2017. New Google algorithm restricts access to left-wing, progressive web sites. Available at: https://www.wsws.org/en/articles/2017/07/27/goog-j27.html.

Damon, A. & North, D., 2017. Google's new search protocol is restricting access to 13 leading socialist, progressive and anti-war web sites. Available at: https://www.wsws.org/en/articles/2017/08/02/pers-a02.html.

Davide, 2015. "MaximumRecords" returns a random subset of results. Available at: http://mantis.tokeek.de/view.php?id=610. Also available at: https://web.archive.org/web/20160309000910/http://mantis.tokeek.de/view.php?id=610.

Douceur, J.R., 2002. The Sybil attack. *IPTPS*, pp.251–260. Available at: https://sci-hub.tw/10.1007/3-540-45748-8_24.

DuckDuckGo, 2012. DuckDuckGo privacy. Available at: https://3g2upl4pq6kufc4m.onion/privacy.

DuckDuckGo, 2018a. DuckDuckGo – GitHub. Available at: https://github.com/duckduckgo.

DuckDuckGo, 2018b. DuckDuckGo – privacy, simplified. Available at: https://3g2upl4pq6kufc4m.onion/.

dumblob, 2018. Mysql2sqlite. Available at: https://github.com/dumblob/mysql2sqlite.

Durham, V., Henkh, K., Sindeyev, M., Kraft, D., Phelix, Castellucci, R., Rand, J., Roberts, B., Bisch, J., Colosimo, A., Cassini, Conrad, P. & Anonymous, 2018. Namecoin. Available at: https://www.namecoin.org/.

Epstein, R., 2015. How Google could rig the 2016 election. Available at: https://www.politico.com/magazine/story/2015/08/how-google-could-

rig-the-2016-election-121548.

Fitblip, 2012. Proxying DNS with python. Available at: https://web.archive.org/web/20161211104525/https://fitblip.pub/2012/11/13/proxying-dns-with-python/.

Free Software Foundation, 2017a. Proprietary tyrants. Available at: https://www.gnu.org/proprietary/proprietary-tyrants.html.

Free Software Foundation, 2017b. What is free software? Available at: https://www.gnu.org/philosophy/free-sw.en.html.

Gerard, 2017. Get domain name from URL. Available at: https://stackoverflow.com/q/9626535.

Google, 2018a. Google. Available at: https://www.google.com/.

Google, 2018b. How Google search works | crawling & indexing. Available at: https://www.google.com/search/howsearchworks/crawling-indexing/.

Google, 2018c. Protocol buffers | Google developers. Available at: https://developers.google.com/protocol-buffers/.

Higgins, S., 2014. 3 pre-Bitcoin virtual currencies that bit the dust. Available at: https://www.coindesk.com/3-pre-bitcoin-virtual-currencies-bit-dust/.

Higgins, T., 2016. How an Eric Schmidt-backed startup may help Clinton get elected. Available at: https://www.bloomberg.com/news/articles/2016-05-19/clinton-bets-on-tech-strategy-to-defeat-trump.

Horist, P., 2017. Lovetox comment on "Gajim with Tor leaks DNS requests". Available at: https://dev.gajim.org/gajim/gajim/issues/8538#note_180861.

Jedusor, T.E., 2016. Mimblewimble. Available at: https://scalingbitcoin.org/papers/mimblewimble.txt.

Joachims, T., Granka, L., Pan, B., Hembrooke, H. & Gay, G., 2005. Accurately interpreting clickthrough data as implicit feedback. *SIGIR*, pp.154–161. Available at: https://sci-hub.tw/10.1145/1076034.1076063.

Jordan, A., Timoshenko, Y. & Maclennan, C., 2018. Consider removing YaCy. Available at: https://github.com/nylira/prism-break/issues/1888.

Kaminsky, D., 2011. Black ops of TCP/IP 2011. Available at: https://media.ccc.de/v/cccamp11-4555-black_ops_of_tcpip_2011-en.

Lin, L., 2016. Linkerlin comment on "support an upstream SOCKS proxy". Avail-

able at: https://github.com/yacy/yacy_search_server/issues/84#issuecomment-258079209.

Lutjohann, D.S., Shah, A.H., Christen, M.P., Richter, F., Knese, K. & Liebel., U., 2011. Sciencenet - towards a global search and share engine for all scientific knowledge. *Bioinformatics*, 27(12), pp.1734–1735. Available at: https://academic.oup.com/bioinformatics/article/27/12/1734/255451.

Masnick, M., 2005. Since when is it illegal to just mention a trademark online? Available at: https://www.techdirt.com/articles/20050105/0132239.shtml.

Masnick, M., 2018. Censorship by weaponizing free speech: Rethinking how the marketplace of ideas works. Available at: https://www.techdirt.com/articles/20180124/11124039076/censorship-weaponizing-free-speech-rethinking-how-marketplace-ideas-works.shtml.

Maymounkov, P. & Mazires, D., 2002. Kademlia: A peer-to-peer information system based on the xor metric. *IPTPS*, pp.53–65. Available at: https://sci-hub.tw/10.1007/3-540-45748-8_5.

Michie, D., 1963. Experiments on the mechanization of game-learning part I. characterization of the model and its parameters. *The Computer Journal*, 6(3), pp.232–236. Available at: https://sci-hub.tw/10.1093/comjnl/6.3.232.

Microsoft, 2018. Bing. Available at: https://www.bing.com/.

Mills, C., 2014. Fwd: 2016 thoughts. Available at: https://wikileaks.org/podesta-emails/emailid/37262.

Mozilla MDN contributors, 2017. Native messaging. Available at: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Native_messaging.

Mozilla MDN contributors, 2018. Browser extensions. Available at: https://developer.mozilla.org/en-US/Add-ons/WebExtensions.

Nakamoto, S., 2009. Bitcoin: A peer-to-peer electronic cash system. Available at: https://bitcoin.org/bitcoin.pdf.

National Security Agency, Snowden, E. & Washington Post, 2013. NSA slides explain the PRISM data-collection program. Available at: https://www.washingtonpost.com/wp-srv/special/politics/prism-collection-documents/.

Niklas B, 2012. Niklas B answer to "parse a tuple from a string?" Available at: https://stackoverflow.com/a/9763133.

pabouk, 2013. Pabouk answer to "how can I test an application for proxy leaks?"

Available at: https://tor.stackexchange.com/a/118.

Parry, R., 2017. The dawn of an Orwellian future. Available at: https://consortiumnews.com/2017/07/28/the-dawn-of-an-orwellian-future/.

Pash, A., 2011. How many times do you search Google every day? Available at: https://lifehacker.com/5743053/how-many-times-do-you-search-google-every-day.

Pass, G., Chowdhury, A. & Torgeson, C., 2006. A picture of search. *The First International Conference on Scalable Information Systems*. Available at: http://uj3wazyk5u4hnvtk.onion/torrent/3511333/MSSQL_AOL_SEARCH_DATA. Also available at: magnet:?xt=urn:btih:7299915525f664e1945f424509f6083d247d0637&dn=MSSQL+AOL+SEARCH+DATA&tr=udp%3A%2F%2Ftracker.leechers-paradise.org%3A6969&tr=udp%3A%2F%2Fzer0day.ch%3A1337&tr=udp%3A%2F%2Fopen.demonii.com%3A1337&tr=udp%3A%2F%2Ftracker.coppersurfer.tk%3A6969&tr=udp%3A%2F%2Fexodus.desync.com%3A6969.

Perry, M., 2011. Tor browser should set SOCKS username for a request based on first party domain. Available at: https://trac.torproject.org/projects/tor/ticket/3455.

Perry, M., 2013. Why the web of trust sucks. Available at: https://lists.torproject.org/pipermail/tor-talk/2013-September/030235.html. Also available at: https://cryptome.org/2013/09/web-trust-sucks.htm.

Perry, M., 2016. The trouble with CloudFlare. Available at: https://blog.torproject.org/trouble-cloudflare.

Podesta, J., 2014. Re: Launch assessment. Available at: https://wikileaks.org/podesta-emails/emailid/41853.

PRISM Break contributors, 2018. Project inclusion guidelines. Available at: https://github.com/nylira/prism-break/blob/bbe535f05adb47dbd0b253ecd2c36e3043e40a8e/README.md#user-content-project-inclusion-guidelines.

Prunster, B., 2018. NetLayer. Available at: https://github.com/JesusMcCloud/netlayer.

Rand, J., 2015. YaCyIndexerGreasemonkey. Available at: https://github.com/JeremyRand/YaCyIndexerGreasemonkey.

Rand, J., 2016a. [DuckDuckGoBridge] add ability to sort by relevance instead of date. Available at: https://github.com/RSS-Bridge/rss-bridge/pull/422.

Rand, J., 2016b. Fix multiple boost queries. Available at: https://github.com/y

acy/yacy_search_server/pull/51.

Rand, J., 2016c. Multiple boost queries are broken in YaCy. Available at: https://gist.github.com/JeremyRand/39d0d77407d651539c7bb920b668d91e.

Rand, J., 2016d. Ranking transparency. Available at: https://github.com/yacy/yacy_search_server/pull/77.

Rand, J., 2016e. Show ranking in HTML UI. Available at: https://github.com/yacy/yacy_search_server/pull/78.

Rand, J., 2016f. Tradeoff between sorting accuracy and display latency. Available at: https://github.com/yacy/yacy_search_server/pull/91.

Rand, J., 2017a. Javascript re-sorting. Available at: https://github.com/yacy/yacy_search_server/pull/104.

Rand, J., 2017b. Tor stream isolation. Available at: https://github.com/bisq-network/bisq-desktop/issues/731.

Rand, J., 2017c. Use MediaWiki external links for Solr boosting. Available at: https://github.com/JeremyRand/wiki-links-popularity-solr.

Rowling, J., 2015. The Marauder's Map. Available at: https://www.pottermore.com/writing-by-jk-rowling/the-marauders-map.

RSS-Bridge contributors, 2018. RSS-Bridge: The RSS feed for websites missing it. Available at: https://github.com/RSS-Bridge/rss-bridge.

Seltzer, W., 2018. Lumen. Available at: https://www.lumendatabase.org/.

ShadowRanger, 2017. ShadowRanger answer to "efficiently count word frequencies in python". Available at: https://stackoverflow.com/a/35857833.

Singhal, A., 2015. A flawed elections conspiracy theory. Available at: https://www.politico.com/magazine/story/2015/08/google-2016-election-121766.

Smith, A., 2014. 6 new facts about Facebook. Available at: https://www.pewresearch.org/fact-tank/2014/02/03/6-new-facts-about-facebook/.

Sommer, E., 2017. Google censors block access to CounterPunch and other progressive sites. Available at: https://www.counterpunch.org/2017/08/09/google-censors-block-access-to-counterpunch-and-other-progressive-sites/.

StartPage, 2018. PRIVACY - how we protect you. Available at: https://www.st

artpage.com/eng/protect-privacy.html.

strace contributors, 2018. Strace: Linux syscall tracer. Available at: https://strace.io/.

Sweatte, N. & Damon, A., 2017. Google turning into "censorship engine" – reporter. Available at: https://www.youtube.com/watch?v=07unP-i--xQ.

Tauber, A., 2018. Welcome to searx. Available at: https://asciimoo.github.io/searx/.

The Apache Software Foundation, 2016. The ExternalFileField type. Available at: https://archive.apache.org/dist/lucene/solr/ref-guide/apache-solr-ref-guide-5.5.pdf#page=59&zoom=auto,0,650.29.

The Apache Software Foundation, 2018a. Apache Lucene. Available at: https://lucene.apache.org/.

The Apache Software Foundation, 2018b. Apache Solr. Available at: https://lucene.apache.org/solr/.

The Tor Project, 2018a. Over the river and through the wood. Available at: https://stem.torproject.org/tutorials/over_the_river.html.

The Tor Project, 2018b. Tor project. Available at: https://www.torproject.org/.

The Tor Project, 2018c. Tor: Onion service protocol. Available at: https://www.torproject.org/docs/onion-services.html.en.

Tor Wiki contributors, 2017. Tor friendly applications best practices: Network configuration. Available at: https://trac.torproject.org/projects/tor/wiki/doc/Tor_friendly_applications_best_practices?version=25#networkconfiguration.

Van der Sar, E., 2016. Google removed over 900 million "pirate" links in 2016. Available at: https://torrentfreak.com/google-removed-over-900-million-pirate-links-in-2016-161230/.

von Ahn, L., Blum, M., Hopper, N.J. & Langford, J., 2003. CAPTCHA: Using hard AI problems for security. *EUROCRYPT*, pp.294–311. Available at: https://sci-hub.tw/10.1007/3-540-39200-9_18.

Whonix contributors, 2018a. Stream isolation. Available at: https://www.whonix

.org/wiki/Stream_Isolation.

Whonix contributors, 2018b. Whonix. Available at: https://www.whonix.org/.

Wikimedia Foundation, 2017. License information about Wikimedia dump downloads. Available at: https://dumps.wikimedia.org/legal.html.

Wikimedia Foundation, 2018. Wikimedia downloads. Available at: https://dumps.wikimedia.org/backup-index-bydb.html.

Wikipedia contributors, 2017a. A&M Records, Inc. v. Napster, Inc. — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=A%26M_Records,_Inc._v._Napster,_Inc.&oldid=796877062. [Online; accessed 10-April-2018].

Wikipedia contributors, 2017b. Constant folding — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Constant_folding&oldid=812501881. [Online; accessed 29-January-2018].

Wikipedia contributors, 2017c. FAROO — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=FAROO&oldid=798128220. [Online; accessed 13-April-2018].

Wikipedia contributors, 2017d. OpenSearch — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=OpenSearch&oldid=816092955. [Online; accessed 15-April-2018].

Wikipedia contributors, 2018a. BitTorrent — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=BitTorrent&oldid=835501278. [Online; accessed 10-April-2018].

Wikipedia contributors, 2018b. Byzantine Generals' Problem — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Byzantine_fault_tolerance&oldid=836194569#Byzantine_Generals%27_Problem. [Online; accessed 15-April-2018].

Wikipedia contributors, 2018c. CAPTCHA — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=CAPTCHA&oldid=835701799. [Online; accessed 15-April-2018].

Wikipedia contributors, 2018d. Cypherpunk — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Cypherpunk&oldid=835641646. [Online; accessed 15-April-2018].

Wikipedia contributors, 2018e. Distributed hash table — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Distr

ibuted_hash_table&oldid=833076603. [Online; accessed 15-April-2018].

Wikipedia contributors, 2018f. Hashcash — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Hashcash&oldid=828549114. [Online; accessed 5-April-2018].

Wikipedia contributors, 2018g. Sybil attack — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Sybil_attack&oldid=833940724. [Online; accessed 5-April-2018].

Wikipedia contributors, 2018h. Web of trust — Wikipedia, the free encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Web_of_trust&oldid=822514577. [Online; accessed 15-April-2018].

YaCy Wiki contributors, 2014. FAQ: How many links/words and how much disk space can a YaCy instance manage? Available at: http://www.yacy-websearch.net/wiki/index.php/En:FAQ#How_many_links.2Fwords_and_how_much_disk_space_can_a_YaCy_instance_manage.3F. Also available at: https://web.archive.org/web/20171221170131/https://www.yacy-websearch.net/wiki/index.php/En:FAQ#How_many_links.2Fwords_and_how_much_disk_space_can_a_YaCy_instance_manage.3F.

YaCy Wiki contributors, 2016. API. Available at: http://www.yacy-websuche.de/wiki/index.php/Dev:API. Also available at: https://web.archive.org/web/20170903065625/http://www.yacy-websuche.de/wiki/index.php/Dev:API.

Yandex, 2018. Yandex. Available at: https://www.yandex.com/.

Zimmermann, P., 1994. How does PGP keep track of which keys are valid? Available at: https://web.pa.msu.edu/reference/pgpdoc1.html#section-7.7.

Appendix 1: Bug report on YaCy's random sorting

This is the verbatim text of the bug report we filed (Rand, 2016f) with YaCy regarding YaCy's sorting appearing to be random.

At the moment, from what I can tell, YaCy waits a very short period of time, then drains exactly 1 result from the Solr and RWI queues into the output queue, and then checks whether the output queue has at least 1 result in it that hasn't yet been displayed to the user. Naturally, this is true 100% of the time. When YaCy sees this, it writes the best result in its output queue (i.e. either the single Solr result or the single RWI result) to the user. It then repeats. (See the section of code modified by this PR to see what code I'm talking about.)

The effect is that there is almost no correlation between the ordering of the results displayed to the user and the ranking calculated by YaCy; the dominant factor is which results got returned first by a peer. Kind of useless as a sorting mechanism.

The easy hack to fix this is to add a significant delay right before the above algorithm runs, which gives the Solr and RWI queues enough time to receive a lot of results before they start being drained into the output queue. The example code in this PR is a hacky way of doing this.

However, the problem is that it adds a long delay before the end user sees any results at all, and there is a tradeoff between the delay and the sorting quality.

My suggestions for a better way to implement a fix than this PR:

1. For API-based use cases that retrieve a single list, e.g. the XML and JSON output formats, provide a way for the user to choose how long to wait for results before the queues start being drained (probably an HTTP GET parameter is

reasonable for this). My guess is that 5 to 10 seconds is going to be enough time for an API user on a fast residential Internet connection (or at least, the first 100 results are likely to be properly sorted in such cases). Users on Tor are likely to need more time.

2. For human-user use cases where Javascript is available, load each result via AJAX and use Javascript to insert it in the correct order. Don't stop receiving results over AJAX when the result limit is hit; keep receiving them for approximately 1 minute. Hide the least relevant results via Javascript once more results have been received over AJAX than the result limit.
3. For API-based use cases where the user is willing to call the API multiple times, re-sort the result list each time the API is called, so that the user gets more relevant results after waiting but they get some possibly-relevant results quickly. Probably one way to do this with minimal code changes is to create a new queue, drain the entire output queue into the new queue, and then replace the output queue with the new queue.

If these suggestions are okay with you, I'm willing to try my hand at implementing them. Or, if you have a better suggestion for how this should be handled, please feel free to let me know.

Appendix 2: Bug report on YaCy's handling of multiple Boost Queries

This is the verbatim text of the bug report we filed (Rand, 2016c) with YaCy regarding YaCy's incorrect handling of multiple Boost Queries.

So I noticed that Boost Queries in YaCy seemed to be producing scores that didn't quite seem right. As a result, I decided to tinker around and see if I could identify what was going on.

Note: this investigation was performed on a self-compiled yacy_v1.83_20160416_9769.

First off, I wanted to know exactly what Solr request YaCy generates when a Boost Query is in use. So:

1. In YaCy's "Ranking and Heuristics" page, reset all of the Solr settings to defaults. This will yield a Boost Query of `crawldepth_i:0^0.8 crawldepth_i:1^0.4` and some Solr Boosts, but no Boost Function or Filter Query.
2. Restart YaCy.
3. Start up Wireshark and have it start capturing.
4. Search with YaCy for something. My query was "Obama".
5. Stop Wireshark's capture.
6. Set the following filter: `http.request.uri contains "solr"`

This yields HTTP request URI's such as the following:

```
1 /solr/select?defType=edismax&qf=url_paths_sxt%5E3.0+synonyms_sxt%5E0
    .5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+h1_txt%5E5.0+
    url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5E2.0+
```

```
description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
=*%2Cscore&q=%22obama%22&bq=crawldepth_i%3A0%5E0.8+crawldepth_i%3
A1%5E0.4&hl=true&hl.fragsize=220&hl.simple.post=%3C%2Fb%3E&hl.
simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.fl=
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&partitions=64&wt=javabin&version=2
```

Okay, so let's try that search locally; go to the following URL:

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
  E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
  h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
  E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
  httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
  url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
  mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
  facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
  Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
```

```
Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
=*%2Cscore&q=%22obama%22&bq=crawldepth_i%3A0%5E0.8+crawldepth_i%3
A1%5E0.4&hl=true&hl.fragsize=220&hl.simple.post=%3C%2Fb%3E&hl.
simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.fl=
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&partitions=64&wt=javabin&version=2
```

Well, that's no good, it asks us to download a .BIN file. Let's change the URL slightly by removing the `&wt=javabin`:

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
=*%2Cscore&q=%22obama%22&bq=crawldepth_i%3A0%5E0.8+crawldepth_i%3
A1%5E0.4&hl=true&hl.fragsize=220&hl.simple.post=%3C%2Fb%3E&hl.
simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.fl=
```

```
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&partitions=64&version=2
```

Whoops, looks like the `&version=2` has to be removed too.

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
  E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
  h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
  E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
  httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
  url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
  mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
  facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
  Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
  Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
  Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
  =*%2Cscore&q=%22obama%22&bq=crawldepth_i%3A0%5E0.8+crawldepth_i%3
  A1%5E0.4&hl=true&hl.fragsize=220&hl.simple.post=%3C%2Fb%3E&hl.
  simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.fl=
  description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
  h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
  h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
  description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
  h1_txt&hl.fl=text_t&partitions=64
```

Okay great, we're getting XML output from Solr. Since we're interested in

ranking calculations, let's enable explanation data output by adding `&debugQuery=true&debug.explain.structured=true` to the query.

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
   E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
   h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
   E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
   httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
   url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
   mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
   facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
   Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
   Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
   Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
   =*%2Cscore&q=%22obama%22&bq=crawldepth_i%3A0%5E0.8+crawldepth_i%3
   A1%5E0.4&hl=true&hl.fragsize=220&hl.simple.post=%3C%2Fb%3E&hl.
   simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.fl=
   description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
   h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
   h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
   description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
   h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
   structured=true
```

Now, let's search through the contents of the `explain` section for some useful tidbits. For example, let's search for `text_t` (which is one of the Solr Boost fields that was enabled). We'll get a bunch of `description` lines similar to `weight(text_t :obama in 60) [ClassicSimilarity], result of:`. Cool, so that tells us that

Solr is indeed processing Solr Boosts (and we could even audit the math if we wanted to).

Now we're going to search for `crawldepth_i`, which is the field used twice in our Boost Query. Hmm, how odd. No matches inside the `explain` section, but there are matches in the `response` section, some of which have the value of 0 or 1 (which should satisfy the Boost Query we set).

So, a reasonable question to ask is, does the Boost Query actually do anything, given that it isn't showing up in the `explain` section? Let's simplify our Boost Query a bit, and compare 2 different values to see how scores are affected.

URL 1:

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
    E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
    h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
    E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
    httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
    url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
    mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
    facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
    Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
    Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
    Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
    =*%2Cscore&q=%22obama%22&hl=true&hl.fragsize=220&hl.simple.post
    =%3C%2Fb%3E&hl.simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.
    fl=description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
    h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
    h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
    description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
```

```
h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
structured=true&bq=crawldepth_i%3A0%5E1.0
```

URL 2:

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
  E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
  h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
  E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
  httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
  url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
  mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
  facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
  Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
  Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
  Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
  =*%2Cscore&q=%22obama%22&hl=true&hl.fragsize=220&hl.simple.post
  =%3C%2Fb%3E&hl.simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.
  fl=description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
  h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
  h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
  description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
  h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
  structured=true&bq=crawldepth_i%3A0%5E2.0
```

As you can see, we've moved the `bq` parameter to the end of the URL (so that it's easier to read at a glance), we've removed the :1 query (so the Boost Query only matches with a crawl depth of 0), and we've setup two URL's where one boosts by 1.0 and the other by 2.0.

Now, let's try both URL's, and look at the explain data for a specific result that has a crawl depth of 0 (should match), and another that has a crawl depth of 3 (shouldn't match).

Hmm, that's interesting. The result with a crawl depth of 0 had its score increase from `1.5181346` to `1.6104686` when the boost was increased from 1.0 to 2.0. The result that has a crawl depth of 3 isn't even in the top 10 results for either search, although it was present before we changed the `bq` parameter. Just to check what's going on, let's increase the row count to 100 so that we can see more results that might not have ranked as highly. These will take a while to load, but be patient (and make sure you have a lot of RAM).

URL 1:

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
    E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
    h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
    E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=100&fq=
    httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
    url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
    mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
    facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
    Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
    Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
    Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
    =*%2Cscore&q=%22obama%22&hl=true&hl.fragsize=220&hl.simple.post
    =%3C%2Fb%3E&hl.simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.
    fl=description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
    h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
    h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
```

```
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
structured=true&bq=crawldepth_i%3A0%5E1.0
```

URL 2:

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=100&fq=
httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
=*%2Cscore&q=%22obama%22&hl=true&hl.fragsize=220&hl.simple.post
=%3C%2Fb%3E&hl.simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.
fl=description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
structured=true&bq=crawldepth_i%3A0%5E2.0
```

So, let's look at two results, one with a crawl depth of 0 and another with a crawl depth of 3. The result with a crawl depth of 0 still increases from `1.5181346` to `1.6104686` just like before. Interestingly, the result with a crawl depth of 3 *de-*

creased from `1.4213018` to `1.4173425`. That's interesting that the 3 result actually decreased rather than staying constant, but at least the 0 result seems to have a clear advantage over the 3 result as a result of the Boost Query.

Hmm, wait a minute, something else is odd here. The 0 result has a node in its `explain` data with the name `weight(crawldepth_i:‘#8;#0;#0;#0;#0; in 63) [ ClassicSimilarity], result of:`. This node has a value of `0.096832804` for the `1.0` boost, and `0.19312611` for the `2.0` boost. That didn't show up anywhere before we tinkered with the `bq` parameter in the URL.

Let's also look at the `boost_queries` section of the `debug` data from before we messed with the `bq` parameter. Its contents are:

```
<arr name="boost_queries">
    <str>crawldepth_i:0^0.8 crawldepth_i:1^0.4</str>
</arr>
```

This is interesting, because both queries are in the same `<str>` element. Why is this interesting? Look right below it, at the `filter_queries` section.

```
<arr name="filter_queries">
    <str>httpstatus_i:200</str>
    <str>-content_type:(image/*)</str>
    <str>-url_file_ext_s:(jpg OR png OR gif)</str>
</arr>
```

Why the discrepancy? And how did we wind up with multiple `<str>` elements for the `filter_queries` section? What's the relevant portion of the URL?

```
&fq=httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
   url_file_ext_s%3A%28jpg+OR+png+OR+gif%29
```

So here we have multiple GET parameters for `fq`, each of which has one query, instead of a single GET parameter for `bq`, which has multiple `+`-separated queries.

Hmm, I wonder what happens if you make `bq` look like `fq`? Let's try it. This is modified from before we tinkered with `bq`. Even the placement in the URL is unmodified (we didn't stick it at the end) so as not to trigger many variables.

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
    E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
    h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
    E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=10&fq=
    httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
    url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
    mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
    facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
    Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
    Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
    Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
    =*%2Cscore&q=%22obama%22&bq=crawldepth_i%3A0%5E0.8&bq=
    crawldepth_i%3A1%5E0.4&hl=true&hl.fragsize=220&hl.simple.post=%3C
    %2Fb%3E&hl.simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.fl=
    description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
    h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
    h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
    description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
    h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
    structured=true
```

Hey cool! The `explain` data now includes lines for the crawl depth! Furthermore, `boost_queries` now shows:

```
1 <arr name="boost_queries">
```

```
    <str>crawldepth_i:0^0.8</str>
    <str>crawldepth_i:1^0.4</str>
</arr>
```

Which seems to match the form of `filter_queries`.

Let's verify that they're working as intended now. We'll set a crawl depth of 3 to have boost `100.0`, and a crawl depth of 1 to have boost `50.0`. Furthermore, we'll move the `bq` parameters back to the end of the URL to see if they're working properly now. We'll also set the row count back to 100 so that we can see past all the 3 results.

```
http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
    E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
    h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
    E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=100&fq=
    httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
    url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
    mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
    facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
    Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
    Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
    Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
    =*%2Cscore&q=%22obama%22&hl=true&hl.fragsize=220&hl.simple.post
    =%3C%2Fb%3E&hl.simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.
    fl=description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
    h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
    h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
    description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
```

```
h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
structured=true&bq=crawldepth_i%3A0%5E50.0&bq=crawldepth_i%3A3%5
E100.0
```

Excellent! The top of the results list is all results that have a crawl depth of 3. The remainder have a crawl depth of 0. (100 rows wasn't enough to get past the 0's.) And the `explain` data shows the correct calculations for all of them.

Now, just to prove to ourselves that YaCy's default usage is broken in terms of ranking order rather than just the `explain` data being wrong, let's run the previous test again, but with the `bq` paramaters collapsed into a single `+`-separated `bq`.

```
1 http://localhost:8090/solr/select?defType=edismax&qf=url_paths_sxt%5
  E3.0+synonyms_sxt%5E0.5+title%5E5.0+text_t%5E1.0+host_s%5E6.0+
  h1_txt%5E5.0+url_file_name_tokens_t%5E4.0+h2_txt%5E3.0+keywords%5
  E2.0+description_txt%5E1.5+author%5E1.0&start=0&rows=100&fq=
  httpstatus_i%3A200&fq=-content_type%3A%28image%2F*%29&fq=-
  url_file_ext_s%3A%28jpg+OR+png+OR+gif%29&facet=true&facet.
  mincount=1&facet.limit=10000&facet.sort=count&facet.method=enum&
  facet.field=%7B%21ex%3Dhost_s%7Dhost_s&facet.field=%7B%21ex%3
  Dauthor_sxt%7Dauthor_sxt&facet.field=%7B%21ex%3Durl_file_ext_s%7
  Durl_file_ext_s&facet.field=%7B%21ex%3Durl_protocol_s%7
  Durl_protocol_s&facet.field=%7B%21ex%3Dlanguage_s%7Dlanguage_s&fl
  =*%2Cscore&q=%22obama%22&hl=true&hl.fragsize=220&hl.simple.post
  =%3C%2Fb%3E&hl.simple.pre=%3Cb%3E&hl.snippets=5&hl.fl=text_t&hl.
  fl=description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
  h1_txt&hl.fl=text_t&hl.fl=description_txt&hl.fl=h4_txt&hl.fl=
  h3_txt&hl.fl=h2_txt&hl.fl=h1_txt&hl.fl=text_t&hl.fl=
  description_txt&hl.fl=h4_txt&hl.fl=h3_txt&hl.fl=h2_txt&hl.fl=
```

```
h1_txt&hl.fl=text_t&partitions=64&debugQuery=true&debug.explain.
structured=true&bq=crawldepth_i%3A0%5E50.0+crawldepth_i%3A3%5E100
.0
```

Yep, it's pretty much a random mixing of crawl depths. Not at all what's intended.

So, what have we learned?

- YaCy's handling of Boost Queries is completely broken whenever you want to have more than one Boost Query.
- As a result, YaCy's default ranking rule of boosting by crawl depth (which is a quite reasonable idea) doesn't work as intended.
- The *wrong* way to handle this (what YaCy does) is to separate Boost Queries with the `+` character in the URL.
- The *right* way to handle this is to supply a separate `bq` GET parameter *for each query*.
- If this is fixed in YaCy, users who upgrade can get improved ranking results, regardless of whether the nodes serving Solr results upgrade.
- This is probably easy to fix in YaCy.

Appendix 3: Etymology of *Heteronculous*

What's with the name?

We decided to name our proxy leak detector *Heteronculous*, and gave it the slogan *For when Phidelius and Mimblewimble just won't save you from Rattus Animagus.* Quoting the "Etymology" section of the Heteronculous README (**spoiler alert: don't read the below etymology if you haven't finished the *Harry Potter* Series!**):

- In the *Harry Potter* series, Peter Pettigrew is the villain responsible for leaking the location of Lily, James, and Harry Potter, resulting in the murder of Lily and James. This is very much akin to proxy leaks resulting in the murder of Muggle activists.
- Pettigrew's role in the leak was discovered using the Marauder's Map (Rowling, 2015), which is implemented via the *Homonculous Charm*.
- "Homonculous" is based on a Latin word that loosely translates to "tiny artificial human"; "homo" is Latin for "human". However, "homo" is also Greek for "same". A proxy leak detector's primary role is to make sure that multiple identities remain separated, so naturally it makes sense to replace "homo" (same) with "hetero" (Greek for "different"), which gives us *Heteronculous*.
- Any complaints from fundamentalist linguists about the mixing of Latin and Greek will be met with a Bat-Bogey Hex.

What's up with the slogan?

- Fidelius is akin to Whonix. It protected the Potters from simple attacks (e.g. Voldemort going looking for them, or a network router detecting your public IP address), but it spectacularly failed to protect them from more esoteric attacks (e.g. Wormtail being a spy, or correlation of identities routed through a pseudonymizing transproxy).
- Mimblewimble is akin to code patches to fix proxy leaks. The Potters didn't think to use it, because they didn't know Wormtail was meeting with Voldemort. (Had the Order of the Phoenix used the Homonculous Charm on Wormtail, perhaps they would have figured it out.) Similarly, you can't patch code that you don't know is leaking; Heteronculous lets you know that your code is leaking so that you can fix it before it leaks your secret location.
- Coincidentally, Phidelius (Kaminsky, 2011) and Mimblewimble (Jedusor, 2016) are the names of existing cryptography projects. It seems that bad Harry Potter jokes are quite popular among cypherpunks.

Appendix 4: Ranking Features Used For Learning (Full List)

The following ranking components were subject to learning (component descriptions quoted verbatim from YaCy's `/IndexSchema_p.html?core=collection1`):

- `sku`: url of document
- `dates_in_content_dts`: if date expressions can be found in the content, these dates are listed here as date objects in order of the appearances
- `startDates_dts`: content of itemprop attributes with content='startDate'
- `endDates_dts`: content of itemprop attributes with content='endDate'
- `title`: content of title tag
- `publisher_t`: the name of the publisher of the document
- `author`: content of author-tag
- `description_txt`: content of description-tag(s)
- `keywords`: content of keywords tag; words are separated by space
- `text_t`: all visible text
- `synonyms_sxt`: additional synonyms to the words in the text
- `h1_txt`: h1 header
- `h2_txt`: h2 header
- `h3_txt`: h3 header
- `h4_txt`: h4 header
- `h5_txt`: h5 header
- `h6_txt`: h6 header
- `inboundlinks_urlstub_sxt`: internal links, the url only without the protocol

- `inboundlinks_anchortext_txt`: internal links, the visible anchor text
- `outboundlinks_urlstub_sxt`: external links, the url only without the protocol
- `outboundlinks_anchortext_txt`: external links, the visible anchor text
- `icons_urlstub_sxt`: all icon links without the protocol and '://'
- `images_text_t`: all text/words appearing in image alt texts or the tokenized url
- `images_urlstub_sxt`: all image links without the protocol and '://'
- `images_alt_sxt`: all image link alt tag
- `bold_txt`: all texts inside of `<b>` or `<strong>` tags. no doubles. listed in the order of number of occurrences in decreasing order
- `italic_txt`: all texts inside of `<i>` tags. no doubles. listed in the order of number of occurrences in decreasing order
- `underline_txt`: all texts inside of `<u>` tags. no doubles. listed in the order of number of occurrences in decreasing order
- `url_file_name_s`: the file name (which is the string after the last '/' and before the query part from '?' on) without the file extension
- `url_file_name_tokens_t`: tokens generated from `url_file_name_s` which can be used for better matching and result boosting
- `url_file_ext_s`: the file name extension
- `url_paths_sxt`: all path elements in the url hpath (see: http://www.ietf.org/rfc/rfc1738.txt) without the file name
- `host_s`: host of the url
- `host_organization_s`: either the second level domain or, if a ccSLD is used, the third level domain
- URL length: internal YaCy field
- Category Image Appearance: a higher ranking level prefers documents with

embedded images

- Category Audio Appearance: a higher ranking level prefers documents with embedded links to audio content
- Category Video Appearance: a higher ranking level prefers documents with embedded links to video files
- Category App, Appearance: a higher ranking level prefers documents with embedded links to applications
- Citation Rank: the more incoming links and the less outgoing links the better the ranking.
- Application Of Prefer Pattern (URL): a higher ranking level prefers documents where the url matches the prefer pattern given in a search request.
- Application Of Prefer Pattern (Title): a higher ranking level prefers documents where the url matches the prefer pattern given in a search request.
- URL Component Appears In Toplist: a higher ranking level prefers documents with words in the url path that match words in the toplist. The toplist is generated dynamically from the search results using a statistic of the most used words. The toplist is a top-10 list of the most used words in URLs and document titles.
- Description Comp. Appears In Toplist: a higher ranking level prefers documents with words in the document description that match words in the toplist. The toplist is generated dynamically from the search results using a statistic of the most used words. The toplist is a top-10 list of the most used words in URLs and document titles.
- Appearance In URL: a higher ranking level prefers documents with urls that match the search word
- Appearance In Title: a higher ranking level prefers documents with titles that match the search word

- `http_unique_b`: unique-field which is true when an url appears the first time. If the same url which was http then appears as https (or vice versa) then the field is false
- `www_unique_b`: unique-field which is true when an url appears the first time. If the same url within the subdomain www then appears without that subdomain (or vice versa) then the field is false
- `title_unique_b`: flag shows if title is unique within all indexable documents of the same host with status code 200; if yes and another document appears with same title, the unique-flag is set to false
- `exact_signature_unique_b`: flag shows if `exact_signature_l` is unique at the time of document creation, used for double-check during search
- `fuzzy_signature_unique_b`: flag shows if `fuzzy_signature_l` is unique at the time of document creation, used for double-check during search
- `description_unique_b`: flag shows if description is unique within all indexable documents of the same host with status code 200; if yes and another document appears with same description, the unique-flag is set to false
- `canonical_equal_sku_b`: flag shows if the url in canonical_t is equal to sku
- `flash_b`: flag that shows if a swf file is linked
- `url_protocol_s`: the protocol of the url (options "http", "https", "ftp", "ftps")
- `crawldepth_i`: crawl depth of web page according to the number of steps that the crawler did to get to this document; if the crawl was started at a root document, then this is equal to the clickdepth

Appendix 5: Google Search Count History Crowdsourced Data (Full List)

This is the full set of users' replies to Adam Pash's Lifehacker article, indicating how many Google searches each user has performed over time.

- Regularly over 76 per day (article author); median 75.5 based on comment about how chart is created
- Lowest 21 per day, Highest 190 per day, 9000-9999 between October 2009 and January 26 2011
- 20688 Between 2007-2008 and January 26 2011
- Over 150 per day on Yahoo
- 20.5 per day
- 45 per day
- 46 so far today
- Lowest this month: 29 per day; Highest this month: 202 per day
- 18 per day
- a few 1-5 days, several in the 20-ish range, a fair smattering of 30s and 40s, and, on 1/11/2011, 178 Googles
- 40000 from account creation to January 26 2011
- 2253 in January 2011; 23932 total searches
- Less than 1 per day
- Around 10 per day
- 41+ per day on work days; weekends significantly lower
- 13.2 per day (between May 26 2007 and January 26 2011)

- 967+3200+4071+3634+2642+3172+985 total (by day of week)
- Median 20 per day
- 22.1 per day in 2 weeks preceding January 26 2011
- Total 19822; Mean 53 per day
- Mean around 80 or 90 per day
- Median 20.5
- 51000 between 2005 and January 26 2011; Mean probably about 60-100 these days
- 10876 total; Mean 5.59 per day lifetime; Mean 9.44 in year preceding January 26 2011
- Around 10 per day
- Total 13948, Mean about 41+
- 151+ every day of 2011 preceding January 26 2011
- Over 150 Yahoo per day
- About 40 per day
- A couple per day
- Around 41 per day, over 70 on weekends
- Minimum 24 per day; Maximum 570 per day; Average over 60 per day at least
- Maybe once per day
- Total 9806
- Total 41049