Web-scale Content Reuse Detection (extended)

USC/ISI Technical Report ISI-TR-692, June 2014

Calvin Ardi John Heidemann

USC/Information Sciences Institute, Marina del Rey, CA 90292

ABSTRACT

With the vast amount of accessible, online content, it is not surprising that unscrupulous entities "borrow" from the web to provide filler for advertisements, link farms, and spam and make a quick profit. Our insight is that cryptographic hashing and fingerprinting can efficiently identify content reuse for web-size corpora. We develop two related algorithms, one to automatically *discover* previously unknown duplicate content in the web, and the second to *detect* copies of discovered or manually identified content in the web. Our detection can also *bad neighborhoods*, clusters of pages where copied content is frequent. We verify our approach with controlled experiments with two large datasets: a Common Crawl subset the web, and a copy of Geocities, an older set of user-provided web content. We then demonstrate that we can discover otherwise unknown examples of duplication for spam, and detect both discovered and expert-identified content in these large datasets. Utilizing an original copy of Wikipedia as identified content, we find 40 sites that reuse this content, 86% for commercial benefit.

1. INTRODUCTION

A vast amount of content is online, easily accessible, and widely utilized today. User-generated content fills many sites, sometimes non-commercial like Wikipedia, but more often commercial like Facebook and Yelp, where it supports supports billions of dollars in advertising. However, sometimes unscrupulous entities repackage this content, wrapping their commercial content around this previously published information to make a quick profit.

There are several recent examples of misleading reuse of content. *Content farming* repost copies of Wikipedia or discussion forums to garner revenue from new advertisements, or to fill out link farms that support search-engine "optimization". *E-book content farming* republishes publicly available information as e-books to attract naive purchasers and spam the e-book market. (Tools like Autopilot Kindle Cash can mass-produce dozens of "books" in hours.) *Review spamming* posts paid reviewers that are often fake and use near-duplicate content to boost business rankings. The common thread across these examples is that they gather and republish publicly available information for commercial gain.

Our goal is to find this kind of republishing on the Internet. We develop a new approach to find duplicate of information systematically across the Internet using hashing. A hash function takes arbitrary content input and produces a statistically unique, simple, fixed-length bitstring. We build lists of hashes of all documents (or "chunks", subparts of documents) in web-size corpora, allowing very rapid detection of content reuse. Although minor changes to content results in different hashes, we show that copying can often be identified by similarities across document chunks. While prior work has explored semantic fingerprints [16, 29, 20] and locality-sensitive hashing [19] for approximate matches, our focus is on exploiting cryptographic hashing to avoid false positives, and to explore blind discovery of duplicated content.

Our approach is designed to detect content reuse for the problems we identified earlier. We show that we can both discover previously unknown duplicated content, such as spam in a partial web corpus, and link farming in Geocities, an older dataset of user content (§ 6.3). We also show our ability to detect expert-identified content, using an original and older copy of Wikipedia, in a web corpus. Utilizing an original copy of Wikipedia as identified content, we find 40 sites that reuse this content, 86% for commercial benefit (§ 7).

The contribution of this paper is to show the potential of hash-based search of web-size corpora to detect content duplication. We show that it is possible to *discover* duplicated content through blind search through a full corpus followed by human-assisted pruning. We also describe our system to *detect* labeled content in a web-size corpus. We demonstrate discovery and detection in our system and show bad neighborhood detection's robustness to random document changes, handling 4.4 − 5.4× the average number of chunks per page in changes. We validate our approach with controlled experiments over two datasets: the Common Crawl subset of the web with 2.86 B files, and in Geocities, an older dataset of user content with 26.7 M files. We then demonstrate that we can discover otherwise unknown examples of duplication for spam, and detect both discovered and expert-identified content in these large datasets.

2. PROBLEM STATEMENT

Replicating web content is easy. Some individuals bulk copy high-quality content from Wikipedia or Facebook to overlay advertisements, or to back-fill for link farms. Others reproduce selected content to impersonate high-value sites for phishing. We seek to develop new approaches to address two problems. First, we want to automatically *discover* content that is widely duplicated, or large-scale duplication in a few places. Second, given list of known duplicated content, we want to *detect where* such content is duplicated. We next define these two problems more precisely.

Consider a *corpus* $\mathbf{C}$ of files f. Interesting corpora, such as a crawl of the Internet, are typically far too large to permit manual examination. We assume the corpus consists of semi-structured text; we use minimal understanding of the semantics of the text to break it into chunks $\mathbf{c}_f$. Each file is identified by URLs; we can exploit the hierarchical structure in the path portion of the URL, or treat them as flat space identified only by the sitename portion.

Our first problem is *discovery.* In discovery our goal is to discover a *labeled dataset* $\mathbf{L}$ consisting of content of interest we expect to be copied. The simplest way to determine $\mathbf{L}$ is for an expert to examine $\mathbf{C}$ and manually identify it. Although not possible in general, semi-automated labeling is suitable for some problems (§ 7) where one one can identify content likely to be copied.

Alternatively, we show how to discover $\mathbf{L}$ through a blind process, without external knowledge. We explore this approach to discover content that is widely duplicated in the web (§ 6).

The *detection* process finds targets $\mathbf{T}$ in the corpus $\mathbf{C}$ that duplicate portions of labeled dataset $\mathbf{L}$. In to finding individual files f that show high levels of copying, we also explore how to exploit the hierarchical grouping of documents in C to find *bad neighborhoods* $\mathbf{N}$ where significant amounts of duplication exists.

3. METHODOLOGY

We next describe our general approach to detecting content reuse. Although we have designed the approach for web-like corpora, it also applies to file systems or other corpora containing textual content like news sources.

3.1 Overview

Our general approach is to compute a hash for each data item, then use hashes to find identical objects. In this section we present our workflow and address the discovery and detection phases of our approach.

Collecting the Data:

0. Crawl the web, or use an existing web crawl, and correct acquisition errors (§ 3.4.1).
1. For each file f in corpus $\mathbf{C}$, compute a hash of the whole file f: $H(f)$ **and**
2. Split f into a vector of chunks $\mathbf{c}_f = \{c_{f,1}, \ldots, c_{f,n}\}$ and hash each chunk $H(c_{f,i})$ to from a *chunk hash vector.* (Although we do not use vector order, the same hash value can appear multiple times if there are duplicate paragraphs in the same file.)

Discovery: (§ 3.2)

3. Populate the labeled dataset with files $\mathbf{L}_f$ or chunks $\mathbf{L}_c$ by either:
   (a) *informed* discovery: seeding it with known content *a priori*
   (b) *blind* discovery: (i) identifying the most frequently occurring files or chunks in $\mathbf{C}$ as suspicious, after (ii) discarding known common but benign content (*stop-chunk removal*, § 3.4.2)

Detection: (§ 3.3)

4. *Simple Object Matching*: Given a labeled dataset objects $\mathbf{L}_o$ (where objects are files or chunks), find all matching objects $o \in C$ where $H(o') \in \mathbf{L}_o$. This results in target (suspicious) files and chunks: $\mathbf{T}_f$ and $\mathbf{T}_c$.
5. *Partial Matching*: To identify files containing partial matches, we use the chunk hash vectors compute the ratio of target chunks to total file content:
   $$\text{contains}(\mathbf{L_c}, f) = \frac{|\mathbf{L}_c \cap H(\mathbf{c}_f)|}{|H(\mathbf{c}_f)|}$$
   If contains$(\mathbf{L_c}, f)$ is greater than a threshold, we consider f to be a *partial target* in $\mathbf{T}_p$.
6. *Bad Neighborhood Detection:* Apply stop-chunk removal (§ 3.4.2), then for each neighborhood
   $$N = \{f_{N,1}, f_{N,2}, \ldots, f_{N,n}\}$$
   where the files share a hierarchical relationship, compute the overall ratio of labeled content matches to total content:
   $$\text{badness}(N) = \sum_{\forall n \in N} \frac{\text{contains}(\mathbf{L}_c, n))}{|N|}$$
   If badness(N) is greater than a threshold, we consider N as a bad neighborhood in $\mathbf{T}_N$.
7. *Push detection to distributed crawlers.* (§ 3.6)

The thresholds for partial matching and bad neighborhood detection are configurable; by default we use one standard deviation over the mean and elaborate on choosing a threshold in § 3.2. We next examine the design decisions behind this algorithm in more detail.

3.2 Discovery

Discovery is the process of building a labeled dataset of items we wish to find in the corpus during detection. We can do this with an informed or blind process.

With *informed discovery* (Step 3a), an expert provides labeled content of interest $\mathbf{L}$, perhaps by exploring $\mathbf{C}$ manually, or using external information. As one example, we know that Wikipedia is widely copied, and so we seed $\mathbf{L}$ with a snapshot of Wikipedia (§ 7). As another example, one could seed $\mathbf{L}$ with banking websites to identify phishing sites that reproduce this content.

One can also identify widely reproduced content through a *blind process* (Step 3b) that automatically discovers widely duplicated content. Blind identification is appropriate when an expert is unavailable, or if the source of copying is unknown. Blind identification populates $\mathbf{L}_f$ and $\mathbf{L}_c$ with the most frequently occurring files or chunks in the corpus. We set the discovery threshold depending on the dataset size and the type of object being identified. For example, one would set the discovery threshold to be higher when the dataset size is larger. We looked at the ROC curves and found a trade-off between false positives (FP) and true positives (TP). There was no strong knee in the curve, thus we picked thresholds with a reasonable balance of FP to TP. In our Common Crawl dataset of 40.5B chunks, we set the threshold to 10^5.

Additionally, in our discovery process we expect to find trivial content that is duplicated many times as part of the web publishing process: the empty file, or a chunk consisting of an empty paragraph, or the reject-all `robots.txt` file. These will inevitably show up very often and litter our $\mathbf{L}$: while common, they are not very significant or useful indicators of mass duplication. To make blind identification more useful, we remove this very common but benign content using stop-chunk removal, described in § 3.4.2.

Given a threshold, all objects o in the corpus $\mathbf{C}$ whose number of duplicates exceeds the threshold and are not "stop chunks" are automatically labeled and added to the labeled dataset $\mathbf{L}$:

$$\mathbf{L} := \forall o \in \mathbf{C} : \text{duplicates}(o) > \text{threshold}, o \notin \{\text{stop chunks}\}$$

We next look at properties of the discovery process.

An important property of discovery is that it is *not distributive*—analysis must consider the entire corpus. While parts of discovery are inherently parallelizable by dividing the corpus to various workers, the final data join and addition of objects to the labeled dataset must be done on the corpus at the same time. Given an example threshold of 1000, consider a corpus $\mathbf{C} = \mathbf{C}_1 \cup \mathbf{C}_2$. Consider an object $j = j_1 = j_2$ such that $|j| = |j_1| + |j_2|$: $j_1 \in \mathbf{C}_1$, duplicates$(j_1) = 1000$ and $j_2 \in \mathbf{C}_2$, duplicates$(j_2) = 100$. When evaluated separately, we see that neither object j_1 or j_2 in their respective corpus are discovered. When evaluated together (duplicates$(j_1 \cup j_2) = 1100$), the number of duplicates of j exceeds our threshold and is then discovered. Thus we have shown it is necessary to run the discovery process on the corpus in its entirety to ensure completeness.

Discovery runtime is $O(n \log n)$ and performance on a moderate-size Hadoop cluster is reasonable (hours). We look at the runtime performance to understand which part of discovery dominates the computation time and, if possible, identify areas for improvement. After we hash all the desired objects ($O(n)$), we sort and count all hashes ($O(n \log n)$), and cull objects ($O(n)$) whose number of duplicates do not exceed the threshold. Discovery's performance is dominated by sorting, leading to an overall performance of $O(n \log n)$.

3.3 Detection

In the detection phase, we find our targets $\mathbf{T}$ at varying levels of granularity in the corpus $\mathbf{C}$ by looking for matches with our labeled dataset $\mathbf{L}$.

In *simple object matching*, our targets $\mathbf{T}$ are an exact match of an object o in $\mathbf{L}$. Given $\mathbf{L}_o$ (where o can be a file f or a chunk c), find all $o' \in \mathbf{C}$ where $H(o') \in \mathbf{L}_o$ and add to the set of targets $\mathbf{T}$. Simple matching is essentially performing a join operation between $\mathbf{L}_o$ (small) and $\mathbf{C}$ (large). Assuming $\mathbf{L}_o$ is constant size, the join is bounded by sorting performance and thus is $O(n \log n)$. We can then analyze $\mathbf{T}$ to understand if objects in $\mathbf{L}$ are being duplicated in $\mathbf{C}$ and how often it is being duplicated. While those statistics are relevant, we expect that duplication happens often and would like to further understand the details and trends of where duplication happens.

We also consider *partial file matching*. Rather than look at whole objects, we can detect target files that partially duplicate content from elsewhere based on a number of bad chunks. Partial matches are files that belong in $\mathbf{T}_p$ because they *contain* part of $\mathbf{L}$. Containment examines the chunk hash vector $H(\mathbf{c}_f)$ of each file to see what fraction of chunks are in $\mathbf{L}$.

Finally, we use *bad neighborhood detection* to look beyond identification of individual files. Examination of "related" files allows detection of regions where large numbers of related files each have a duplicated copy. For example, finding a copy of many Wikipedia pages might lead to a link farm which utilized Wikipedia to boost its credibility or search engine ranking.

We define a neighborhood based on the hierarchical relationship of files in the corpus. A neighborhood N is defined by the URL prefix p, it consists of all files $f \in \mathbf{C}$ where $p(f) = p(N)$.

Many sites have relatively shallow hierarchies, so in the worst case each site is a neighborhood. However, for complex sites with rich user content (as in our Geocities dataset), individuals may create distinct neighborhoods. Each site will have neighborhoods at each level of the hierarchy. For `arxiv.org/archive/physics/`, we would consider three neighborhoods: `arxiv.org/archive/physics/`, `arxiv.org/archive/`, and `arxiv.org/`.

We assess the quality of a neighborhood by applying partial matching to all chunks in the neighborhood N using contains($\mathbf{L_c}, N$) in Step 5 and add N to the set of targets $\mathbf{T}$ if the result is greater than a threshold. Like chunk hash vector for files, the neighborhood chunk hash vector will have duplicated components when there are multiple copies of the same chunk in the neighborhood. Because neighborhood analysis is done over a larger sample, when we find regions that exceed our detection threshold, it is less likely to represent an outlier and instead show a set of files with suspicious content. We next look at properties of the detection process.

Unlike discovery, the detection process is parallelizable when processing *distinct* neighborhoods N (i.e., neighborhoods that do not share the same URL prefix). This parallelizable property allows us to process many neighborhoods simultaneously without affecting whether a particular neighborhood is detected as "bad" or not. Given $\mathbf{C}_1$ and $\mathbf{C}_2$, we assert that detected($\mathbf{L}, \mathbf{C}_1 \cup \mathbf{C}_2$) = detected($\mathbf{L}, \mathbf{C}_1$) $\cup$ detected($\mathbf{L}, \mathbf{C}_2$). This holds true because $\mathbf{C}_1$ and $\mathbf{C}_2$ share no neighborhoods: given some neighborhood $N \in \mathbf{C}_1$, $N \notin \mathbf{C}_2$. As before, runtime performance is $O(n \log n)$ because of the sort during join. However, since neighborhoods are independent and numerous, we get "easy" parallelism. With p processors, we get runtime $O(n \log n)/p$.

3.4 Cleaning the Data

We do two types of cleaning over the data, first we identify recursion errors that result in false duplication from the crawling process, and then we eliminate common, benign features with *stop-chunk removal.* We evaluate the effectiveness of these methods in § 5.1.

3.4.1 Detecting and Handling Recursion Errors

Crawling the real-world web is a perilous process, with malformed HTML, crawler traps, and other well understood problems [6, 17]. We detect and remove crawler artifacts that appear in both Common Crawl and Geocities. Our main concern is recursion errors, where a loop in the web graph duplicates files with multiple URLs—such results will skew our detection of copied data. We see this problem in both datasets and use heuristics involving how often a URL path component is repeated and remove that URL from processing if it is determined to be a recursion error. We evaluate these heuristics in § 5.1, finding that these heuristics have a very low false positive rate in detecting crawler problems, and are sufficient to avoid false positives in our duplication detection. Future work may refine these heuristics to reduce the number of false negatives in recursion-error removal.

3.4.2 Stop Chunk Removal

We see many common idioms in both files and chunks. We call these *stop chunks*, analogous to stop words in natural language processing (such as "a", "and", and "the"). For chunks, these include the empty paragraph (`<p></p>`), or a single-space paragraph (`<p>&nbsp;</p>`). For files, examples are the empty file, or a reject-all `robots.txt` file. These kind of common, benign idioms risk skewing our results.

We remove stop chunks before applying bad neighborhood detection. We maintain a list of 226 stop chunks. This list is short enough to allow manual comparison; if it goes too large we can apply Bloom filters to support efficient stop-chunk removal [5]. We have developed this list manually. Potential future work is to automate stop-chunk discovery.

3.5 Choice of Hash Function

Central to our work is choice of a hash function, which will reduce arbitrary data to a short value. We employ the SHA-1 [21, 13] cryptographic hash function for its precision—identical input always produces the same output, and different input yields a different output. Alternatives such as locality-sensitive [19] and semantic [24] hashes are also possible, but potentially introduce a large number of false positives that would occur in a corpus the size of the web. We go into more detail about our choice of hash function and why we prefer it over other alternatives in the technical report [4]. Explicit comparison to locality-sensitive and semantic hashing algorithms is an opportunity for future work.

3.6 Shifting Detection Into Network

Although our prototype evaluates web data centrally, it can improve crawling and distributed detection.

Our hash-based detection algorithm allows efficient distributed detection of content, given a labeled dataset. Hashing's efficient detection can be placed in distributed crawlers, or at copy-detection tools at hosting sites. Pushing detection into crawlers or sites improves efficiency, avoiding the need to centralize content for detection. In addition, our studies about detection robustness to mutation apply to distributed detection.

Our blind discovery algorithm can generate this labeled dataset foreknowledge of what is being copied.

4. DATASETS AND IMPLEMENTATION

Datasets: This paper uses two public web datasets: Common Crawl and Geocities. We use the Common Crawl `crawl-002` dataset ($\mathbf{C}_{cc}$) collected in 2009/2010 and publicly provided by the Common Crawl Foundation [10] to represent recent web data. `crawl-002` includes 2.86 B items, which is 26 TB compressed and 99 TB uncompressed. Most of its data is HTML or plain text, with some supporting textual material (CSS, JavaScript, etc.); it omits images.

As a second dataset, we use the Geocities archive ($\mathbf{C}_g$) taken the Archive Team [3] just before the Geocities service was shuttered by Yahoo! in October 2009. The dataset was compiled between April–October 2009 and contains around 33M files (650 GB compressed) including documents, images, MIDI files, etc. in various languages. Although this dataset is considerably older, it provides a relatively complete snapshot of user-generated generated content.

Implementation: We implement our methods and post-processing on a local cluster of 55 commodity PCs running Apache Hadoop [1]. Processing was done with custom MapReduce programs [11], Apache Pig [2], and GNU Parallel [28]. Our current cluster can intake data at a rate around 9 TB/hour.

Common Crawl data is stored in and publicly available on Amazon S3. Our initial hashing of files and chunks in $\mathbf{C}_{cc}$ was done using AWS in 11.5k compute hours, producing 2.86 B file hashes and 40.5 B chunk hashes along with backreferences to the original dataset (about 1.8 TB of metadata). Discovery and detection are done over these datasets in our local cluster.

5. VALIDATION

We next validate our design choices, showing the importance of cleaning and correctness of our methodology.

5.1 Do Our Cleaning Methods Work?

Initial analysis of our raw data is skewed by downloading and ripping errors, and identification of bad neighborhoods can be obscured by common benign content. We next show that our cleaning methods from § 3.4 are effective.

We have reviewed our data and taken steps to confirm that recursion errors do not skew discovery of duplicates. While only 1% of all 913M neighborhoods in Common Crawl are the result of recursion errors, removing the obvious errors is helpful although not essential. Details of recursion error removal and its validation are omitted here due to space, but are in our technical report [4].

We next describe validation that our stop-chunk removal process (§ 3.4.2) is effective. To identify stop chunks, we manually examine the 500 most frequently occurring chunks in Common Crawl and identify 226 as benign. These chunks occur very frequently in the dataset as a whole, accounting for 35% of all chunks that occur $\geq 10^5$ times. To verify that we do not need to consider additional frequent words, we also examine the next 200 and identify only 43 as benign, showing diminishing returns (these 43 account for only 1% of all

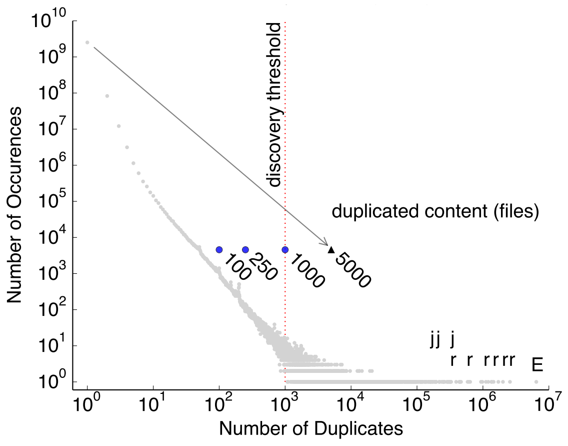

Figure 1: File-level discovery of injected duplicates (black triangle) in $\mathbf{C}_{cc}$, compared to file frequency (grey dots). Very common files: j: JavaScript, r: `robots.txt`, E: the empty file.

chunks that occur $\geq 10^5$ times). We therefore stop with the benign list of 226 chunks found in the top 500 most frequent as it is sufficient to avoid false positives due to benign data.

To demonstrate the importance of common listing, we compare bad neighborhood detection with and without common listing. Stop chunks dilute some pages; if we leave stop chunks in the Common Crawl dataset, we detect 1.88 M (2.35%) more bad neighborhoods than the 79.9 M bad neighborhoods we find after stop-chunk removal, compared to 900 M total neighborhoods.

5.2 Can We Discover Known Files and Chunks?

We next turn to the correctness of our approach. We begin by validating that hashing can detect specific content in spite of the background "noise" of millions of web pages with the following experiment.

Duplicated full files: We first consider a spammer that duplicates a file many times to provide content for thousands of parked domains. To emulate this scenario, we take a known web page (`blog.archive.org` as of 2013-08-22) and duplicate it from $d = 100$ to 5000 times. For each duplication we generate a unique, emulated website, process that data with steps 0–2 of our methodology, merging this with our full processed data.

We then build our labeled dataset via blind discovery. Our blind discovery process populates the labeled dataset with the most frequently occurring content. In Common Crawl ($\mathbf{C}_{cc}$), our blind discovery threshold is 10^3: all files that have more than 10^3 duplicates are labeled.

Figure 1 shows the results of this experiment in $\mathbf{C}_{cc}$ file frequency. Our discovery threshold is marked by a red dotted line at $x = 10^3$; all the content (indicated by points) past the threshold are added to the labeled dataset. Duplicating the blog moves it from unique, unduplicated content (a grey dot in the top left) to an outlying point with 4k pages occurring 5000 times (as indicated by a black triangle labeled 5000). We see that the point passes our threshold and we have discovered our injected and massively duplicated blog. This change from top-left to an outlier above and further right on the graph represents what happens when spammers duplicate portions of the web.

Spammers may duplicate files fewer number of times. To consider this scenario, we change the number of duplications d to values less than our previous example. The blue circles represents the injected file had it been duplicated different amounts of times (at $d = 100$, 250, 1000). When the injected files have been duplicated $\leq 10^3$ times (three blue circles on and to the left of the red threshold line), those files will not be automatically discovered; all points past the red threshold line (the black triangle at $x = 5000$) will. Note that even with fewer duplications, the injected files duplicated fewer times will still be obvious outliers on the graph and may be detected with manual analysis or with more sensitive automation.

Partially duplicated pages: The above experiment shows our ability to track duplicated files, but spammers almost always add to the duplicated content to place their own links or advertisements. We therefore repeat our study of duplicating files, but add a different paragraph to the head and tail the duplicated page to represent unique advertisements attached to each page. Since each page varies here, file-level analysis will detect nothing unusual, but chunk-level analysis will show outliers. The size distribution of pages is skewed and appears heavy tailed (with mean of 48 chunks, median of 23, and max 428). Our discovery threshold is increased from 10^3 to 10^5, because the number of chunks in $\mathbf{C}_{cc}$ is much larger than the number of pages.

Figure 2 shows chunk-level valuation of this scenario. The red dotted line at $x = 10^5$ marks our discovery threshold: all 6k chunks past the line are discovered, added to the labeled dataset, and further analyzed.

We now see more evidence of duplicated chunked content, which is shown by a cluster of black triangles (as opposed to a single outlying point) corresponding to the 1.2 B chunks that make up the duplicated content of `blog.archive.org` (originally 242 K chunks). The light grey circles correspond to all the existing chunks in $\mathbf{C}_{cc}$.

We see that many of the chunks that make up the pages of `blog.archive.org` pass our defined threshold and we discover 78% of total distinct chunks. Similar to the previous experiment, we can "control" where the points are distributed by varying the number of times we duplicate the site. If all the chunks in the site had fallen below the threshold, we would not have automatically discovered the site via our blind discovery process.

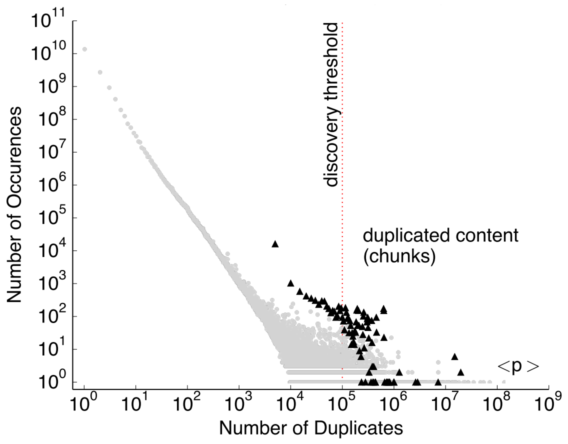

Figure 2: Chunk-level discovery of injected duplicates (black triangles and blue circles) in $\mathbf{C}_{cc}$, compared to chunk distribution (grey dots).

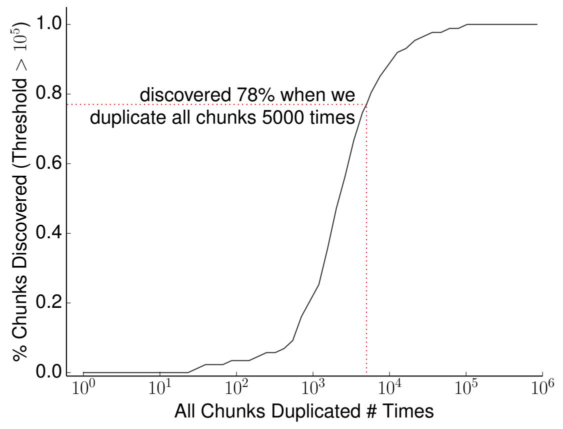

Figure 3: Percentage of chunks discovered in `blog.archive.org` given the number of times `blog.archive.org` is duplicated.

Hashing a finer-grained object in our discovery process allows us to discover more content that has been duplicated. File-level discovery returns a binary result: either we discover the file or not. Chunk-level discovery allows us to discover varying percentages of content depending on how many times it was duplicated. Figure 3 shows how many chunks from `blog.archive.org` are discovered given the number of times all chunks have been duplicated. When we duplicate all the chunks 5000 times (black triangles in Figure 2 and the point marked by the red dotted line in Figure 3), we discover 78% of the chunks. (Trivially, we discover 100% of the chunks when we duplicate the site $\geq 10^5$ times.)

Our simple threshold detects some but not all duplicated chunks that were injected. The duplicated content (black triangles) in Figure 2 are clear outliers from most of the traditional content (grey dots), suggesting a role for manual examination. This experiment shows that chunk-level analysis is effective even though only portions of pages change. We next look at the effects of content mutation more systematically.

5.3 Can We Detect Specific Bad Pages?

Having shown that we can discover known files and chunks, we next validate our detection mechanism by finding known targets $\mathbf{T}$ and understanding the conditions in which our mechanism fails. Given our labeled dataset curated by an expert ($\mathbf{L}_{\text{expert}}$) and one via blind discovery ($\mathbf{L}_{\text{blind}}$), can we detect bad pages? Furthermore, if we gradually mutate each page, at what point can we no longer detect the mutated page?

To evaluate our bad page detection mechanism, we continue our prior example where we rip and duplicate `blog.archive.org`; this set of pages becomes our injected corpus $\mathbf{C}_i$. We mutate $\mathbf{C}_i$ in a consistent manner that can be applied to all pages in $\mathbf{C}_i$ to get a resulting $\mathbf{C}'_i$. We can categorize each mutation into the following:

$+$ Add additional content, such as ads or link spam

Δ Modify existing content by rewriting links

$-$ Remove content such as headers, copyright notices, footers, or the main body of the the page

We build both $\mathbf{L}_{\text{expert}}$ and $\mathbf{L}_{\text{blind}}$ from $\mathbf{C}_i$ (as described in § 5.2), then run the detection process to see if pages in $\mathbf{C}'_i$ are detected.

We continue mutating $\mathbf{C}'_i$ (e.g., $\mathbf{C}''_i, \ldots, \mathbf{C}'^{(n)}_i$) to understand the kinds and amount of mutations that the detection process can handle. While we utilize a copy of `blog.archive.org` to build $\mathbf{L}$ and $\mathbf{C}_i$, our results for each mutation experiment are consistent with other $\mathbf{L}$ because we mutate each of the 4626 pages. For each experiment, we have the base site $\mathbf{C}_i$ and apply n independent mutations to each page resulting in $\mathbf{C}'^{(n)}_i$.

In our first mutation experiment, we continuously *add* content to a page such that the page is diluted with non-target content and we do not detect it (due to the badness ratio not reaching a particular threshold). Figure 4 shows the performance with both $\mathbf{L}_{\text{expert}}$ (green) and $\mathbf{L}_{\text{blind}}$ (blue). The bottom x-axis details the number of chunks added per page *relative* to the average number of chunks per page ($\overline{\text{cpp}} = 48$). The y-axis shows the average badness ratio per page (averaged over all 4626 pages in $\mathbf{C}_i$). The badness threshold is labeled on each graph at 0.144 (we describe its computation in a later section). We perform 10 runs over $\mathbf{C}_i$ at each x value and take the average. We omit error bars when the standard error is < 0.01 for clarity.

This experiment shows that we can tolerate $3.4\times$ or more the mean number of chunks per page ($\overline{\text{cpp}}$) in each labeled dataset and still detect duplicated content ($\mathbf{L}_{\text{blind}}$: $3.4\times$, $\mathbf{L}_{\text{expert}}$: $4.5\times$). The behavior is not surprising: if we were to dilute the content with many other unrelated chunks, the average badness would asymptotically approach 0.

We next continuously *delete* content randomly; this will increase the badness ratio but may be overlooked because the number of total chunks on the page will be smaller (e.g., a page with a badness ratio of 1.0 containing only 3 chunks seems negligible). Figure 5 shows the average badness of a page given the number of chunks we delete per page. Using an $\mathbf{L}_{\text{expert}}$ (green), we see that the ratio is always 1.0: deleting chunks does not affect the badness because the entire page is bad regardless. Next, while we initially see a increase in badness when using $\mathbf{L}_{\text{blind}}$ (blue), it is not monotonically increasing as the number of deleted chunks per page increases. In this experiment, our detection mechanism on average handles all 400 deletions (per page).

We see a large variance in badness at the tail of the graph because the population of pages in $\mathbf{C}'^{(n)}_i$ (after mutation) decreases; as we increase the number of deleted chunks per page, the average number of chunks per page (orange) falls. Pages also cease to exist after all the chunks have been deleted. This behavior is expected: as a trivial example, consider a page with only two chunks only one of which is in $\mathbf{L}$: the badness of the page is 0.5. If we delete the bad chunk, the badness falls to 0, but if we delete the other, the badness increases to 1. Thus, depending on the chunks we delete, the badness of a page will fluctuate.

In our final experiment, we continuously *modify* content to the point where we no longer can detect it (e.g., if every chunk is modified at least once, our detection algorithm will fail). We consider a *stream* of mutations: we randomly pick a chunk to modify, with replacement (in successive mutations, the same chunk can be modified again). Figure 6 shows the average badness of a page given the number of random changes with replacement. We see an exponential drop in the average badness of the page as we linearly increase the number of random changes (with replacement) per page. On average, our bad page detection mechanism handles $1.8 \times \overline{\text{cpp}}$ ($\mathbf{L}_{\text{blind}}$) and $2.0 \times \overline{\text{cpp}}$ ($\mathbf{L}_{\text{expert}}$) changes before the page falls below the threshold.

To show that we can tolerate $3.4 \times \overline{\text{cpp}}$ mutations, we look at the performance of our bad page detection mechanism. Figure 7 shows how many pages we detect as bad given the number of random changes per page in $\mathbf{C}_i$. In the perfect case (such as using $\mathbf{L}_{\text{expert}}$ on an unmodified site), we detect all 4.6k pages in $\mathbf{C}_i$ as bad. While the $\mathbf{L}_{\text{expert}}$ performs much better initially (detecting between 300-700 more pages than with $\mathbf{L}_{\text{blind}}$), we see both lines eventually converge.

We can detect known bad pages to a certain degree of mutation. Our validation experiments show that we can handle between $1.8 - 4.5 \times \overline{\text{cpp}}$ mutations on $\mathbf{C}_i$ depending on the type of mutation and the labeled dataset

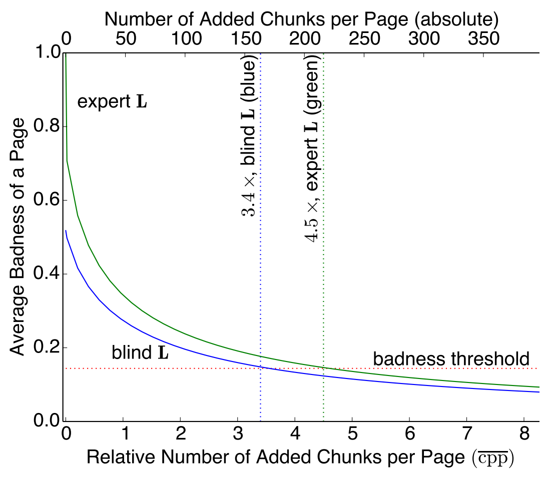

Figure 4: Effects of continuously adding chunks on pages.

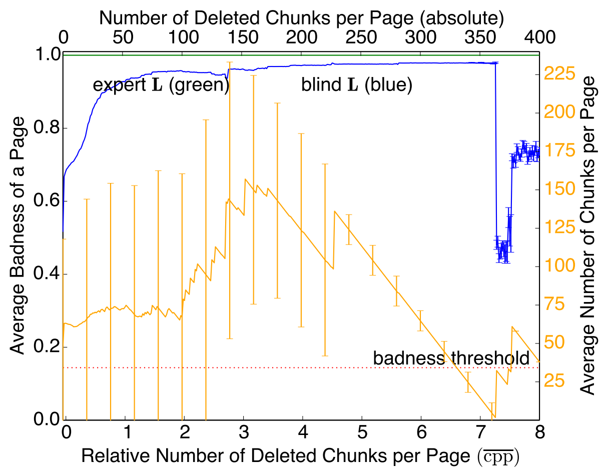

Figure 5: Effects of continuously deleting chunks on pages.

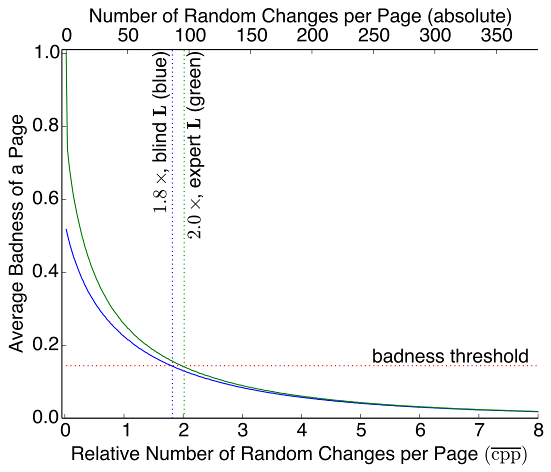

Figure 6: Effects of continuously changing chunks on pages.

we utilize. While utilizing the $\mathbf{L}_{\text{expert}}$ slightly increases the number of mutations we can tolerate (compared to using the $\mathbf{L}_{\text{blind}}$), the $\mathbf{L}_{\text{expert}}$ contains over 4.8× the number of entries ($|\mathbf{L}_{\text{expert}}| = 21\text{k}$, $|\mathbf{L}_{\text{blind}}| = 4.4\text{k}$). We next transition into the validation of detecting known bad neighborhoods.

5.4 Can We Detect Known Bad Neighborhoods?

Given our success finding bad pages, we next validate the robustness of detecting known bad neighborhoods. Recall that a neighborhood contains a set of pages that share a common URL prefix. As with pages, we evaluate both expert and blind labeled datasets, and change a known target to evaluate the sensitivity of our detection mechanism.

We evaluate our detection mechanism by designing a mutation experiment with an example neighborhood N. The goal of our experiment is to understand the degree of change before our detection process fails. We continue to use the same neighborhood N (`blog.archive.org`) and the same approach as in the previous section (§ 5.3) with the following change: mutate all pages in N in a consistent manner to get a resulting N': n mutations results in $N'^{(n)}$. We then run the bad neighborhood detection process to see if $N'^{(n)}$ is detected.

We see similar results in the performance of bad neighborhood detection compared to bad page detection. Figures 8 through 10 show the bad neighborhood detection performance using both $\mathbf{L}_{\text{expert}}$ (green) and $\mathbf{L}_{\text{blind}}$ (blue) for add, delete, and modify operations, respectively. We compare the relative number of mutated chunks per page ($\overline{\text{cpp}}$) against the resulting badness ratio of the neighborhood after mutation ($N'^{(n)}$). We use a fixed badness threshold as described in § 6.3. We again take the average over 10 runs over N at each x

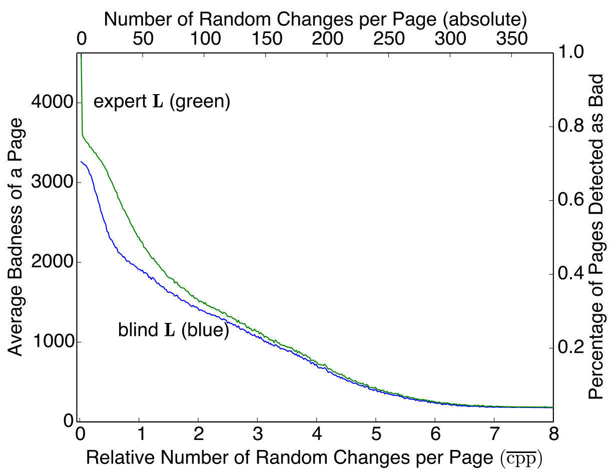

Figure 7: Number of pages detected as bad after continuously changing chunks on pages in `blog.archive.org`.

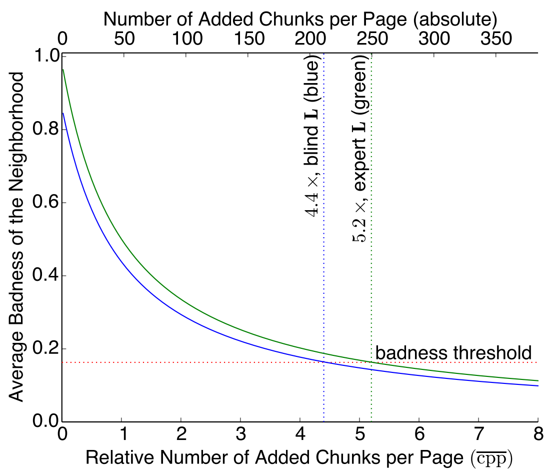

Figure 8: Effects of continuously adding chunks on each page in N.

Description	$|\mathbf{f}|$	subcat. %	**Type**
Common (benign)	45		Benign
`robots.txt`	32	71%	Benign
JavaScript libraries	8	18%	Benign
HTTP error pages	5	11%	Benign
Empty	2		Benign
Misc.	3		Benign
Total	50		

Table 1: Categories of the 50 most frequent files in $\mathbf{C}_{cc}$

value and omit error bars when standard error is < 0.01.

Our experiments show that we can tolerate between $4.4 - 5.4 \times \overline{\text{cpp}}$ mutations, and that bad neighborhood detection is much more robust than bad page detection—on average our process can handle $2.7 - 3.0\times$ more modifications per page than bad page detection. Analysis of the neighborhood is much more robust because we consider the badness across a collection of pages and have a larger population of content to work with; considering only a page when calculating badness is much more susceptible to fluctuation and not as robust to mutation because of its smaller magnitude.

We have now validated our mechanisms that we will now use in two applications: content reuse detection over web content using the blind process and detection of expert-identified content in the web.

6. APPLICATION: DUPLICATION OF WEB CONTENT

We next turn to applying our approach by considering discovery and detection of duplication of web content, applying blind discovery to both our Common Crawl and Geocities datasets. We use this application first to compare design alternatives of file- and chunk-level hashing, then we use our method to evaluate bad neighborhoods in each dataset.

6.1 Why is File-level Discovery Inadequate?

We first consider file-level discovery on both datasets. File-level comparisons are overly sensitive to small changes, so we do not expect to find interesting duplicated content, but to instead establish a baseline against which to evaluate chunk-level comparisons.

Figure 1 shows the frequency of file-level hashes for the 2.86 B files in Common Crawl. The data shows a long-tail distribution of file frequency, as one would expect, with 2.52 B unique files, and 75 distinct files with more than 10^5 duplicates. We label several benign duplicated files, including the empty file (E), several variants of `robots.txt` (r), and several common JavaScript libraries (j). Table 1 describes the top 50 most occurring files; all are benign.

More interesting files might not be in the top 50. Since the top 50 are all benign, we next look at a random sample of frequently occurring files. We draw 40 random files from the 7569 unique files with more than 10^3 occurrences. Table 2 shows how we classify this sample. We see that file-level detection finds only benign content, finding commonly reused infrastructure.

We confirm these results by repeating this process on Geocities ($\mathbf{C}_g$). We find similar prevalence of libraries and benign shared files. The primary difference is that we see 85% of the top 150 most frequent files in Geocities are images, including the Geocities logo, colored bullets, and similar logos.

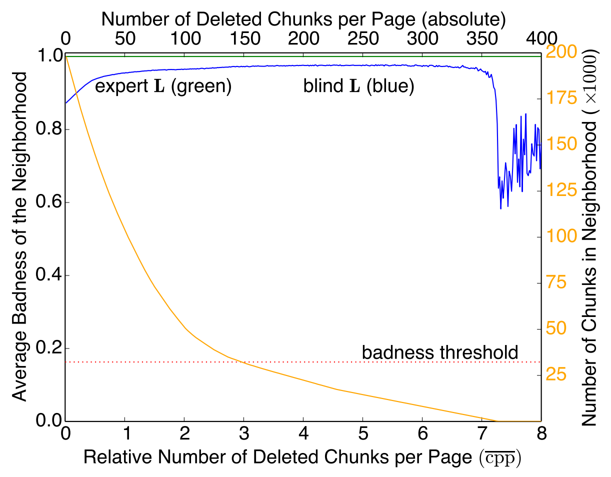

Figure 9: Effects of continuously deleting chunks on each page in N.

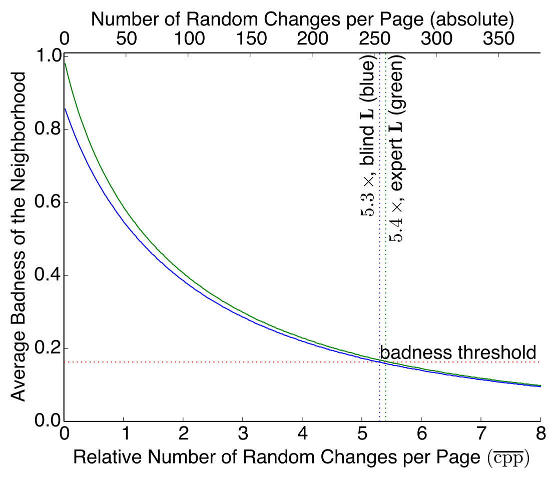

Figure 10: Effects of continuously changing chunks on each page in N.

Description	|f|	subcat. %	Type
Common (benign)	32		Benign
JavaScript libraries	13	41%	Benign
`robots.txt`	9	28%	Benign
HTTP error pages	8	25%	Benign
Cascading Style Sheet	2	6%	Benign
Misc.	8		Benign
Pornography	1	12.5%	Benign
Placeholder	1	12.5%	Benign
Other	6	75%	Benign
Total	40		

Table 2: Classification of a sample of 40 unique files with more than 10^3 occurrences in $\mathbf{C}_{cc}$

Description	|c|	subcat. %	Type
Common (benign)	68		Benign
Templates	17	100%	Benign
e-Commerce	8	47%	Benign
Other	9	53%	Benign
Misc.	15		Benign
Total	100		

Table 3: Categories of the top 100 distinct chunks in $\mathbf{C}_{cc}$.

6.2 How Does Chunking Affect Discovery?

We next turn to chunking to see if duplication of portions of web pages is more common than entire pages. Here we focus on text-based files (HTML, plaintext, JavaScript) and use paragraph-based chunking (§ 3.1).

Figure 2 shows frequency-occurrence distribution of the 40.5 B chunks in Common Crawl ($\mathbf{C}_{cc}$). Again, we see a heavy-tailed distribution, where about 40% of chunks are unique, but about 3.7 B distinct chunks appear more than 10^5 times. The most common chunk is the empty paragraph (`<p>`).

The finer resolution of chunking allows us to begin to see more interesting duplication. Analysis shows discovery of different kinds of duplication: *affiliate links*, JavaScript *ads/tracking* and *scripts*, and *benign content* dominating the list. Table 3 classifies the 100 most frequent chunks. After common web idioms (empty paragraph, etc.), we see templates from software tools or web pages begin to appear.

Description	|c|	subcat. %	Type
Misc.	4		Benign
JavaScript	7	100	Benign
advertising	3	42	Ambiguous
tracking	2	29	Ambiguous
escaped	1	14	Benign
other	1	14	Benign
Templates	83	100	Benign
navigation	17	20	Benign
forms	32	39	Benign
social	4	5	Benign
other	30	36	Benign
Spam	1		Malicious
Spam? / Scam?	5		Ambiguous
Total	100		

Table 4: Classification of a sample of 100 distinct chunks with more than 10^5 occurrences in $\mathbf{C}_{cc}$

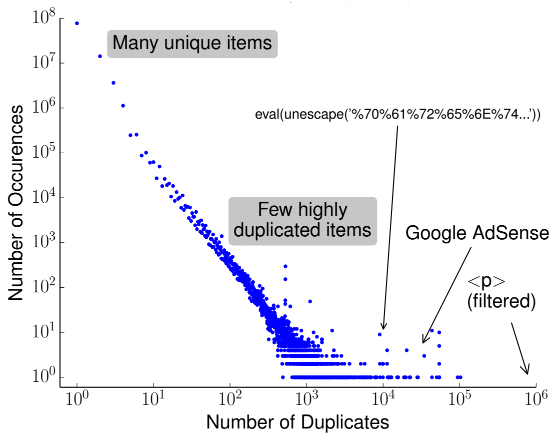

Figure 11: Chunk-level Discovery on $\mathbf{C}_g$ (97M chunks, after heuristic and benign listing).

Again, we turn to a random sample of the tail to understand what makes up duplicated content. We draw a sample of 100 chunks from those with more than 10^5 occurrences and classify them in Table 4.

This sample begins to show common web components that support monetization of websites. JavaScript occurs some (7%) and used for advertising via Google AdSense (3%), user tracking, and analytics (2%). We sampled one instance of spam where an article from The Times (London) was copied and an advertising snippet was included in the article for travel insurance. Other snippets were potentially spam-like or linking to a scam (5%), but ambiguous enough to qualify as a non-malicious (if not poorly designed for legitimate monetization) site.

We also find instances of potentially malicious escaped JavaScript: decoding it reveals an email address (obfuscated via JavaScript to throw off spammers). Most content we discovered are elements of sites that make heavy use of templates (83%) such as navigation elements, headers, and footers. Given an $\mathbf{L}_o$ of the most frequently occurring content, this is not surprising: thousands of pages containing such template elements would naturally show up at the tail of the distribution.

We confirm our results over a second dataset with chunk-level discovery on $\mathbf{C}_g$ (Geocities) in Figure 11. We see a similar distribution overall, and similar kinds of templates and JavaScript as in $\mathbf{C}_{cc}$.

We discovered and examined the kinds of content duplicated in $\mathbf{C}_{cc}$. Chunking identifies frequent duplication, but not bad behavior. However, we can now use the results to build a labeled dataset of objects $\mathbf{L}_o$. We next utilize $\mathbf{L}_o$ in our detection mechanism to identify and detect areas where copying runs rampant.

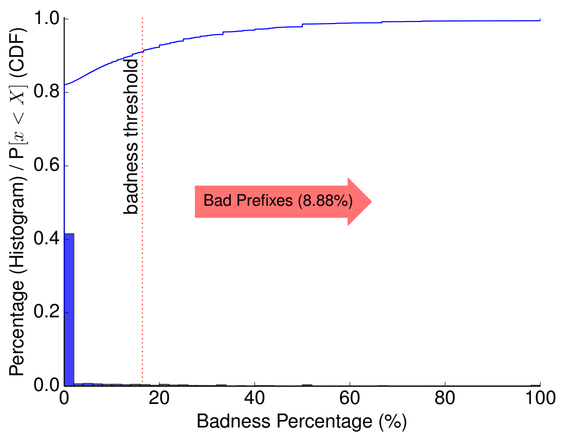

Figure 12: Frequency of badness of neighborhoods in $\mathbf{C}_{cc}$, as a histogram (bars) and CDF (lines).

6.3 Are There Bad Neighborhoods in the Real World?

Chunking is successful at identifying bad chunks and pages, but duplication for profit can draw on many related pages to maximize commercial potential. Detection at the individual page-level can result in false positives, so we would prefer to detect groups of related pages that show a significant amount of copied content. We now shift our focus to detecting bad neighborhoods.

In Common Crawl: To look for bad neighborhoods, we utilize the top 2121 common distinct chunks from Common Crawl as our labeled dataset $\mathbf{L}_c$ (from § 3.2), and identify bad neighborhoods in the full dataset using the algorithm in § 3, step 6. $\mathbf{C}_{cc}$ contains 900 M neighborhoods. Our detection threshold uses the mean and standard deviation across all neighborhoods.

As one would hope, most neighborhoods $N \in \mathbf{C}_{cc}$ are not bad (91%). Figure 12 shows a combined histogram and CDF the bad content ratio of all neighborhoods. We observe that 79.8 M prefixes (9%) out of 900 M would be classified as a bad neighborhood: neighborhoods with badness > 0.163 (since $\mu_{N,cc} = 0.04$ and $\sigma_{N,cc} = 0.123$, and the threshold is $\mu_{N,cc} + \sigma_{N,cc}$).

To understand the nature of the neighborhoods we identify as employing common content, we extract a sample of 40 neighborhoods from the 19.6 M that are above the threshold and classify them in Table 5. We find 82.5% of the sampled sites to be benign: mostly blogs, forums, or newspapers that make heavy use of templates. Only 13% of the content is clearly for profit: either spam, or search-engine optimization with ads.

Our results show that there is duplication on the web: our approach discovers it through a blind process and then detects the broad kinds of copying that exists. Our approach is best at finding content that uses templates or uniform, repeated structure. Most content with this

Description	$\lvert N\rvert$	%
advertisements*	2	5
blog*	19	47.5
empty	1	2.5
forms	1	2.5
forum	1	2.5
"suspect"	0	0
JavaScript	2	5
templated site or CMS	17	42.5
Total*	43	

Table 5: Classification of a sample of 40 neighborhoods drawn with badness above threshold, from $\mathbf{C}_{cc}$. Asterisks (*) indicate overlap between categories.

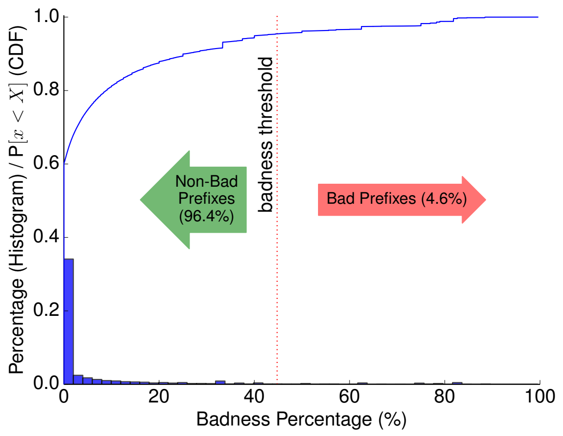

Figure 13: Frequency of badness of neighborhoods in $\mathbf{C}_g$, as a histogram (bars) and CDF (lines).

structure is benign, but we find a fraction of it is spam. The prevalence of templates in the sites we detect is a direct result of obtaining **L** via our blind process, since the definition of **L** is commonly reused content. This observation suggests that a labeled dataset more focused on malicious content (not just duplicated content) would improve the yield, as we explore in § 7 with a expert-provided **L**.

In Geocities: We next evaluate Geocities to confirm our findings from bad neighborhood detection in Common Crawl. Here we consider the top 2121 common distinct chunks from Geocities as our labeled dataset $\mathbf{L}_c$, and identify bad neighborhoods in $\mathbf{C}_g$ using the same method. Figure 13 shows a combined histogram and CDF of bad content ratios and indicate that most of our neighborhoods have low badness ratios. We observe that 36,780 neighborhoods (4.56% of the 806,769 in the dataset) pass the threshold of badness (0.448, since here $\mu_{N,g} = 0.152$ and $\sigma_{N,g} = 0.296$).

In Table 6 we classify a sample of 100 randomly chosen neighborhoods above the badness threshold from Geocities. We find a much higher rate of malicious neighborhoods, with 40% of the sampled appearing to be link farms. These link farms contained lists of self-referential links for credit cards, auto loans, and financing (among other things) with Google AdSense advertisements (now defunct) to generate ad revenue. Another link farm again contained self-referential links with haphazard keywords ranging from automobile manufacturers to downloading MP3s. The common thread tying these bad neighborhoods together was the repeated usage of keywords and links along with the advertisement. The rest of the sampled neighborhoods in our Geocities sample are benign. We find heavy use of the default Geocities templates, as well as other templates.

Description	$\lvert N\rvert$	Type
Link farm	40	Profit
Templates	50	Benign
default	37	Benign
other	13	Benign
Misc	10	Benign
Total	100	

Table 6: Classification of a sample of 100 neighborhoods drawn with badness above threshold, from $\mathbf{C}_g$.

In general, the Geocities results confirm what we find in Common Crawl. We believe the higher rate of copying in link farms reflects the fact that the content dates from several years earlier, a time when search engines were more easily influenced by their input.

7. DETECTION OF EXPERT-IDENTIFIED CONTENT IN THE WEB

We next turn to detecting expert-identified labeled content in the web. This approach is useful when the target is known, but the locations of where it is copied is unknown. We illustrate this approach by looking for copies of Wikipedia on the web. Although Wikipedia's license allows duplication [30], copies add little benefit for users, even if they generate revenue from attached advertisements. More negative uses are copies that exist only to boost search ratings through link farms, or to dilute spam or other content. Another future application is detection of phishing websites: an expert could create a labeled dataset from phishing candidates (banking and e-commerce sites), sites with copies of this content are likely phishing sites.

To evaluate if our approach to detecting content reuse works in practice, we evaluate the detection process with an expert-identified labeled dataset (**L**). We quantify how much of **L** is copied elsewhere and understand the nature of such duplicates: are copies wholesale rips or are snippets of particular articles being cut and pasted? Similarly, for what reasons is content from

Description	$\|N\|$	%	subcat. %	Type
Clones/Rips of Wikipedia	31	78	100	
"Wikipedia Ring"	13		42	Profit
Reference Sites	5		16	Profit
Advertisements	10		32	Profit
Fork	1		3	Ambig.
Unknown	2		6	Ambig.
Wikipedia / Wikimedia	5	13		Benign
Search Engine Optimization	3	8	100	
e-Commerce	2		67	Profit
Stock Pumping	1		33	Profit
Site utilizing MediaWiki	1	3		Benign
Total	40			

Table 7: Classification of the top 40 bad neighborhoods in $\mathbf{C}_{cc}$, $\mathbf{L}$ = Wikipedia.

L being copied?

To evaluate these questions, we utilize a curated labeled dataset **L** and detect targets **T** that copy in part or whole from **L**. We select for **L** a static HTML dump of Wikipedia in English created in June 2008 [15]. Each page of Wikipedia is then processed, resulting in a curated labeled dataset $\mathbf{L}_\mathrm{c}$ of 75.0M distinct chunks of length > 100, and then search for this content in the Common Crawl corpus ($\mathbf{C}_{cc}$). Utilizing $\mathbf{L}_\mathrm{c}$, we identify bad neighborhoods in $\mathbf{C}_{cc}$ using the algorithm described in § 3, step 6.

We expect to find the original Wikipedia and its sister sites (Wiktionary, Wikiquote, etc.) as well as sites that may be unofficial Wikipedia mirrors. We also expect to find partial rips of Wikipedia with advertisements appended and other sites that utilize Wikipedia content to promote a brand, service, or product.

Our detection mechanism finds 136k target neighborhoods (almost 2% of 68.9M neighborhoods in $\mathbf{C}_{cc}$) of path length 1 that include content from Wikipedia. To understand how and why more than 100k sites copy parts of Wikipedia, we focus our analysis on neighborhoods that duplicate from Wikipedia the most. We look at the top 40 neighborhoods with the highest number of bad chunks and classify them in Table 7. We find 5 sites that were directly affiliated with Wikimedia, including Wikipedia, Wikibooks, and Wikisource. More interestingly, we find 34 instances of duplicate content on third party sites: 31 sites rip Wikipedia in a wholesale manner, and the remaining 3 utilize content from Wikipedia in a subtle fashion for search-engine optimization (SEO).

Upon closer examination of the 31 third-party clones of Wikipedia, we found that almost all are using Wikipedia content to promote commercial interests. One interesting example was a "Wikipedia Ring": a group of 13 site rips of Wikipedia, with external links to articles that leads to another site in the ring. In addition to the intra-ring links, each site had an advertisement placed on each page to generate revenue. Other clones exhibited similar behavior, sometimes in conjunction with the addition of other content either scraped from other sources or manually curated.

Duplicated Wikipedia content was also utilized to promote a brand or product. One example site utilized information about retirement plans in the United States to additionally promote a specific stock to purchase (akin to stock promotion email spam). Another site subtly utilized content to increase its rankings in search engines to lure unwary visitors into playing a slot machine game.

Overall we conclude that we are able to discover and detect duplicate content and previously unknown copies of our labeled dataset on the web. Our approach detects target neighborhoods that copy from our labeled dataset curated from Wikipedia, leading us to find that in most cases (34 from our sample of 40), Wikipedia is being utilized for profit. While the Wikipedia license allows duplication in a specific manner, it's clear that various operators are monetizing Wikipedia content without attribution. Similarly, while we detect sites that utilize content from and properly credit Wikipedia, the utility of this content is unclear to anyone else but the operators. This application case study shows the ability of our approach to find such duplication. Future work will entail different applications, including how we can detect previously unknown phishing sites.

8. RELATED WORK

There is significant prior work in detection of duplicated content, to reduce storage or network use, and to find near-duplicate content for plagiarism detection or information retrieval.

Storage and Network Optimization: Content duplication detection serves many purposes and several fields and industries have revolved around the idea. Data de-duplication can be utilized to efficiently store similar or identical pieces of data once instead of multiple times [23]. The same concept can be utilized over the network to reduce the amount of data transferred between two nodes [26, 22]. We build on this work and their analysis of chunking. While they target the application of storage or network optimization, we instead consider detection of duplication for commercial gain. Their application forces efficient, on-line processing and a relatively small corpus, while web analysis is suitable for off-line analysis with corpus sizes of tens to thousands of terabytes.

Plagiarism Detection: Plagiarism detection is a very different class of application. While storage and network optimization requires exact reproduction of original contents, the cost of missing a duplicate is much higher in plagiarism detection, while false duplication can be corrected in review. Existing approaches to plagiarism detection therefore emphasize approximate matching over moderate size corpora [25, 12, 14]. Our work aims to answer the question of whether massive duplication exists on a web-scale using syntactic methods; we do not attempt to infer semantic equivalence of the content.

Information Retrieval: Document similarity and detection is at the heart of the field of information retrieval (IR). Approaches in IR have explored duplicate detection to improve efficiency and the precision of answers [20, 8, 16, 27]. Our use of cryptographic hashing has high precision at the cost of lower recall by missing mutated files.

Broder et al. [7] develop a new technique called "shingling" (today known as n-grams) to cluster documents that are "roughly the same". They use this technique to find documents that are nearly identical in content and potentially located at different URLs to find documents that move. While their focus is on finding syntactically related documents on the web, it can be applied to finding content reuse on the web. One can consider an n-gram as an approximate hash that identifies a document; we instead use cryptographic hashes to provide high precision. We also separate discovery and detection, allowing blind or informed identification of the labeled dataset.

Henzinger [16] compares algorithms that use shingling and Charikar's locality sensitive hashing (LSH) on 1.6 B web-pages to compare their performance. While LSH achieves better precision than shingling, combining the two algorithms provides an even higher precision. Exploration of LSH is an interesting possible complement or addition to our use of cryptographic hashing.

SpotSigs [29] is a robust signature for documents that combines stop-words and short chains of adjacent terms. They also develop a matching algorithm using partitioning and inverted index pruning to search for similarities. SpotSigs' approach is similar to n-grams, while we use a different hashing and clustering method. We also separate the discovery and detection sides of the problem.

Chiu et al. [9] develop a system to detect instances of text reuse on sentence-level queries that are written about the same topic but in different styles. They compare results using Yahoo!'s API, Iterative Chunking, and Query n-grams and find that n-grams works best in their application. Their goal is analogous to the detection phase of our work, where they find a set of documents using approximate matching. We do both discovery and detection with exact matching of chunks.

Yang and Callan [31] develop a system that implements a clustering algorithm using document metadata as constraints to group near-duplicates together in EPA and DOT document collections. They exploit constraints in document metadata; we instead focus on general datasets that provide no such metadata.

Kim et al. [18] develop an algorithm for overlaps and content reuse detection that is both efficient (fast) and incremental (given new content, compare to entries in the data collection). After creating sentence-level signatures for each document, processing a query in the system involved converting it to a signature and returning documents that contain signatures that are some Euclidean distance $\sqrt{d}$ of each other. Their focus is sentence-level reuse in blogs and news articles with approximate matching. We focus on larger chunks of reuse and exact matching.

9. CONCLUSIONS

In this paper we developed a method to *discover* previously unknown duplicated content and to *detect* that or other content in a web-size corpus. We also showed how to exploit hierarchy in the corpus to identify bad neighborhoods, improving robustness to random document changes. We verified that our approach works with controlled experiments, then used it to explore duplication in a recent web crawl with an informed and uninformed discovery process. Although most duplicated content is benign, we show that our approach does detect duplication as-is in link farms and webpage spamming.

10. REFERENCES

[1] Apache. Hadoop. `http://hadoop.apache.org`, 2012.

[2] Apache. Pig. `http://pig.apache.org`, 2013.

[3] ArchiveTeam. Geocities. `http://archiveteam.org/index.php?title=Geocities`, 2009.

[4] C. Ardi and J. Heidemann. Web-scale content reuse detection (extended). Technical Report ISI-TR-692, USC/Information Sciences Institute, June 2014.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, pages 107–117, Brisbane, Queensland, Australia, Apr. 1998.

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[8] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *In Proc. of 34th STOC*, pages 380–388. ACM, 2002.

[9] S. Chiu, I. Uysal, and W. B. Croft. Evaluating text reuse discovery on the web. In *Proceedings of the third symposium on Information interaction*

in context, IIiX '10, pages 299–304, New York, NY, USA, 2010. ACM.

[10] CommonCrawlFoundation. Common crawl. `http://commoncrawl.org`, 2012.

[11] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

[12] H. Dreher. Automatic conceptual analysis for plagiarism detection.

[13] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), Sept. 2001. Updated by RFCs 4634, 6234.

[14] S. M. Z. Eissen and B. Stein. Intrinsic plagiarism detection. In *Proceedings of the European Conference on Information Retrieval ECIR-06*, 2006.

[15] W. Foundation. Static html dump of wikipedia, June 2008.

[16] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 284–291, New York, NY, USA, 2006. ACM.

[17] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World-Wide Web Journal*, 2(4):219–229, Dec. 1999.

[18] J. W. Kim, K. S. Candan, and J. Tatemura. Efficient overlap and content reuse detection in blogs and online news articles. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 81–90, New York, NY, USA, 2009. ACM.

[19] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Data Bases*, pages 950–961, Vienna, Austria, Sept. 2007. VLDB Endowment.

[20] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 141–150, New York, NY, USA, 2007. ACM.

[21] National Institute of Standards and Technology. Secure hash standard (SHS). Federal Information Processing Standard (FIPS) 180-3, National Institute of Science and Technology, Oct. 2008.

[22] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[23] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[24] R. Salakhutdinov and G. Hinton. Semantic hashing. *Int. J. Approx. Reasoning*, 50(7):969–978, July 2009.

[25] A. Si, H. V. Leong, and R. W. H. Lau. Check: a document plagiarism detection system. In *Proceedings of the 1997 ACM symposium on Applied computing*, SAC '97, pages 70–77, New York, NY, USA, 1997. ACM.

[26] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *In Proceedings of ACM SIGCOMM*, pages 87–95, 2000.

[27] B. Stein, M. Koppel, and E. Stamatatos. Plagiarism analysis, authorship identification, and near-duplicate detection PAN'07, Dec. 2007.

[28] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[29] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 563–570, New York, NY, USA, 2008. ACM.

[30] Wikipedia. Reusing wikipedia content, 2014. [Online; accessed 31-Mar-2014].

[31] H. Yang and J. Callan. Near-duplicate detection by instance-level constrained clustering. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 421–428, New York, NY, USA, 2006. ACM.

APPENDIX

A. REMOVING RECURSION ERRORS

When we examine the distribution of the lengths of all neighborhoods in the Common Crawl dataset (Figure 14), we observe a noticeable gap in the distribution when prefix length is 97.

The right of the graph is all recursion errors, all links 97 or longer (0.005% of data) is bad. We confirm this with a random sample of 50 pages of path length 97 or longer. We therefore set a threshold of 96, retaining almost all of the original data. We know that some of this data is still bad: a sample of 100 neighborhoods of length 20 or longer (0.46% of total data) show that about two thirds are bad, but we limit pruning to avoid pruning valid data. As future work, we plan to explore

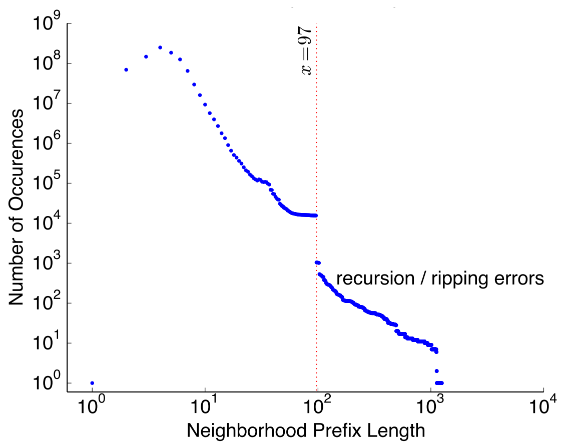

Figure 14: Prefix lengths of neighborhoods in $\mathbf{C}_{cc}$.

better methods to remove ripping errors.

While some recursion errors remain, our cleaning leaves most pages unaffected by recursion errors: a random sample of 100 of all lengths suggests that only 1% of all neighborhoods are recursion errors.

We found similar problems in Geocities, and solved them with a similar cleaning process. While each new data crawl will need to be reviewed for collection errors, resolution of problems is straightforward and additional work will not be required as individual crawlers mature.

B. CHOICE OF HASH FUNCTION

Central to our work is choice of hash function. We employ a cryptographic hash function, but alternatives such as locality-sensitive and semantic hashes are also possible. Each reduces arbitrary data to a short value, but they behave differently in the face of data mutation.

We choose a cryptographic hash for its precision—identical input always produces the same output, and different input yields a different output. A hash collision for any two distinct strings would happen with probability 2^{-b} for a b-bit hash. The drawback to utilizing a cryptographic hash function is that even minor changes to the input give a different output, causing otherwise duplicate content to pass undetected. Our primary method to counter this effect is chunking, so that mutations must occur to every chunk in a file to have it completely escape a match. We could also reduce the impact of minor changes by normalizing the input, perhaps collapsing case and whitespace; evaluation of normalization is future work.

Locality sensitive hashing (LSH) generalizes cryptographic hashing, effectively performing parallel cryptographic hashes to allow detection of similar but non-identical data (for example, [19]). In a sense, chunking is an "application-level" LSH approach; explicit comparison to LSH algorithms is an opportunity for future work.

Semantic hashes does linguistic (or at least syntactic) analysis of the text to compute a hash that represents the meaning of the source material [24]. In the extreme, it can identify two sentences that talk about the same topic without sharing any words in common. The disadvantage of semantic hashing is the large number of false positives that would occur in a corpus the size of the web.

The specific cryptographic hash we use is NIST SHA-1 [21, 13]. It is reasonably resilient to collisions and The SHA-1 hash function takes in arbitrary input and produces a reasonably short, fixed-length, 160-bit hash value. In principle we can use any similar hash algorithm. We considered SHA-2 and SHA-3, but current implementations are significantly slower than the more mature SHA-1.