Precise Detection of Content Reuse in the Web

Calvin Ardi
USC/Information Sciences Institute
calvin@isi.edu

John Heidemann
USC/Information Sciences Institute
johnh@isi.edu

ABSTRACT

With vast amount of content online, it is not surprising that unscrupulous entities "borrow" from the web to provide content for advertisements, link farms, and spam. 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 *precisely detect* copies of discovered or manually identified content. We show that *bad neighborhoods*, clusters of pages where copied content is frequent, help identify copying in the web. We verify our algorithm and its choices with controlled experiments over three web datasets: Common Crawl (2009/10), GeoCities (1990s–2000s), and a phishing corpus (2014). We show that our use of cryptographic hashing is much more precise than alternatives such as locality-sensitive hashing, avoiding the thousands of false-positives that would otherwise occur. We apply our approach in three systems: discovering and detecting duplicated content in the web, searching explicitly for copies of Wikipedia in the web, and detecting phishing sites in a web browser. We show that general copying in the web is often benign (for example, templates), but 6–11% are commercial or possibly commercial. Most copies of Wikipedia (86%) are commercialized (link farming or advertisements). For phishing, we focus on PayPal, detecting 59% of PayPal-phish even without taking on intentional cloaking.

CCS CONCEPTS

• **Information systems → Spam detection**; **Web mining**; *Extraction, transformation and loading*; Data cleaning; • **Security and privacy → Phishing**.

KEYWORDS

content reuse, content duplication, duplicate detection, phishing

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 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* involves reposting 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 reviews 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 develop methods that efficiently find duplication in large corpora, and to show this approach has multiple applications. We show that our method (§7) efficiently finds instances of mass-republishing on the Internet: for example, sites that use Wikipedia content for advertising (§8.1). While copying Wikipedia is explicitly allowed, bulk copying of copyrighted content is not. Even when allowed, content farming does little to enhance the web, and review spamming and e-book content farming degrade the impact of novel reviews and books, much as click fraud degrading legitimate advertising. Our approach can also detect phishing sites that use duplicated content to spoof users (§8.2).

Our insight is that *cryptographic hashing* can provide an effective approach in duplication detection, and scales well to very large datasets. 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 result in different hashes, we show that copying can often be identified in the web by finding the same chunks across documents. Economically, spammers seek the greatest amount of profit with minimal work: we see that current spammers usually do not bother to obfuscate their copying. (If our work forces them to hide, we at least increase their effort.) Our work complements prior work in semantic fingerprints [24, 31, 44] and locality-sensitive hashing [28]. Such approaches provide approximate matching, reducing false negatives at the cost of some false positives. While semantic hashing is ideal for applications as computer forensics, where false positives are manageable, our approach is relevant to duplicate detection in web-size corpora, where *precise* matching without false positives, since even a tiny rate of false positives overwhelms true positives when applied to millions of documents on the web (§6.5). We also explore blind (automated) discovery of duplicated content.

We evaluate our approach on several very large, real-world datasets. We show that *blind discovery* can automatically find previously unknown duplicated content in general web-scale corpora (§7), evaluating Common Crawl (2.86×10^9 files) and GeoCities (26.7×10^6 files). While most general duplication is benign (such as templates), we show that 6–11% of widespread duplication on the web is for commercial gain. We also show that *expert-labeled datasets* can be used with our approach to efficiently search web-size corpora or to quickly search new pages on the web. We demonstrate bulk searches by looking for copies of Wikipedia on the web (§8.1), finding that most copies of Wikipedia (86%) are commercialized (link farming or advertisements). We also show that our approach can detect phishing in web pages (§8.2), demonstrating a Chrome plugin and evaluating it with a targeted dataset that finds that 59% of PayPal phish, even without taking measures to defeat intentional cloaking (for example, source-code obfuscation).

Contributions: The contribution of this paper is to show that hash-based methods can blindly *discover* content duplication then *detect* this duplication in web-size corpora. The novelty of this work is not the use of hashing (a long-existing technique), but design choices in adapting hashing (with chunking and cleaning, §4.4) to scale to discovery (§4.2) and detection (§4.3) across web-size datasets, and to operate robustly in the face of minor changes. In particular, our approach to discovery can be thought of as a form of semi-supervised machine learning. One important step is our use of the hierarchical nature of the web to find clusters of copied content ("bad neighborhoods", §7.3). We show that this approach applies not only to discovering general duplication (§7) and identifying bulk copying in the web (§8.1), but also to detecting phishing activity (§8.2). We support our work with validation of the correctness of discovery and detection in our methodology (§6) and evaluation of bad neighborhood detection's robustness to random document changes (§6.4). Our data and code are available for research reproducibility (§5.3).

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 a 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* **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 $\mathbf{c}_f$ by choosing basic, structural delimiters (without attempting to infer the meaning of the content itself). 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* **L** consisting of content of interest we believe to be copied. The simplest way to determine **L** is for an expert to examine **C** and manually identify it, thus building an *expert-labeled dataset* **L** from content in **C**. (The corpus **C** used to build **L** can be the same as or different than the corpus later used in detection—for now we use the same **C** in discovery and detection.) Although not possible in general, semi-automated labeling is suitable for some problems (§8) where one can build **L** independently from known information.

Alternatively, we show how to discover **L** through a *blind discovery* process, without external knowledge. We explore this approach to discover content that is widely duplicated in the web (§7).

The *detection* process finds targets **T** in the corpus **C** that duplicate portions of labeled dataset **L**. In addition to finding individual files that show high levels of copying, we also exploit the hierarchical grouping of documents in **C** to find *bad neighborhoods* **N**, defined as clusters of content sharing the same URL hierarchy where many files appear to be duplicated.

3 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, and in phish detection.

Storage & Network Optimization: Content duplication detection serves many purposes and several fields have revolved around the idea. Data deduplication can be used to efficiently store similar or identical pieces of data once [30, 35, 47]. Our work shares some of the same fundamental ideas through the use of cryptographic hashing and chunking to effectively find duplicate or similar files. They have explored, for example, chunking files into both fixed- and variable-sized blocks and hashing those chunks to find and suppress duplicate chunks in other files. In our chunking methods, we consider variable-sized blocks delimited by HTML tags, leveraging the structure provided by the markup language. While they target the application of storage optimization of files, focusing on archival hardware and systems, our applications target duplicate detection at both the file- and neighborhood-level for commercial gain.

The same concept can be used in a network to reduce the amount of data transferred between two nodes [34, 41]. Network operators can reduce WAN bandwidth use by hashing transmitted packets (or chunks of packets) in real time and storing this data in a cache. Network users can then see improved download times when retrieving data matched in the cache (for example, when multiple users download the same file or visit the same webpage). Our work also uses the idea of hashing to detect duplicates, but with different applications. While their work looks at suppressing redundant, duplicate downloads from the web, our work looks at finding where duplication exists on the web. Their application forces efficient, streaming processing and a relatively small corpus (caches are less than 100 GB to minimize overhead), while our web analysis is suitable for offline processing with corpora larger than 1 PB.

Plagiarism Detection is a very different class of application. Storage and network optimization requires exact reproduction of original contents. Existing approaches to plagiarism detection in documents emphasize semantic matching, as plagiarism is also concerned with subtle copying concepts, in addition to exact text reuse. Plagiarism detection makes use of stylometric features [19, 40], measuring writing structure and styles, in addition to text statistics (counts of words and parts-of-speech). 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.

Because detecting plagiarism is typically done over small- to moderate-sized corpora (comparing a essay or homework assignment to ~1000 others), long runtimes (minutes to sometimes hours per document [17]), and a relatively large rate of false positives (precision = 0.75, for example [19]) are tolerable. Manual review can address false positives, and with a relatively small corpus, the absolute number of false positives can be manageable even if the rate is not small. In our applications, our parallelized processing enables us to maintain good performance, even as the corpus grows (§5.2). Additionally, we require high precision in detecting reuse since with large corpora (10^9 documents or more), even a small false positive rate quickly makes human review impractical (§6.5).

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 [11, 24, 31, 42]. Our use of cryptographic hashing has high precision at the cost of lower recall by missing mutated files.

Broder et al. [10] develop a technique called "shingling" (today known as *n*-grams) to generate unique contiguous subsequences of tokens in a document and cluster documents that are "roughly the same". They use this technique to find, collapse, and ignore near-duplicates when searching for information (to avoid showing users the same content multiple times). In our applications, we specifically look for content matches and require new approaches (cryptographic hashes) to avoid overwhelmingly numbers of false positives from approximate matching (§6.5).

SpotSigs [44] and Chiu et al. [12] use *n*-grams in different applications to search for similarities and text reuse with approximate matching. SpotSigs extends the use of *n*-grams by creating signatures of only semantic (meaningful) text in documents, ignoring markup language details (like HTML tags). Their system for approximate matching is quadratic ($O(n^2)$) in its worst-case, but it can trade-off runtime performance with threshold of similarity. Our work looks for precise content matches in quasilinear time ($O(n \log n)$). In one of our applications in phish detection, we leverage the details of HTML to enable precise detection of phish.

Chiu et al. build an interface within a web browser to interact with their back-end, enabling users to query for sentence-level similarities between their visited page and other pages in a larger corpus. We use precise matching of paragraph-level chunks, with applications on detecting widespread duplication across the web. We distinguish between the discovery and detection sides of the problem, allowing us to better separate the problems of finding "*what* content is duplicated" and finding "*where* the content is being duplicated". In Chiu et al., "discovery" is a manual search query performed by the user, while in our work we can perform discovery as an automated process.

Cho et al. [13] use sentence-level hashing of text to search for duplicated content in order to improve Google's web crawler and search engine. By identifying duplicates with hashing, a web crawler becomes more efficient as it avoids visiting clusters of similar pages. Because the crawler avoids and suppresses duplicates, the quality of search results is improved with more diverse sites being returned in response to a user's query. Our work complements this prior work with different chunking strategies and different applications in measurements and anti-phishing. While Cho et al. extract, chunk, and hash only textual information (no markup), we look at paragraph-level chunking of a document's native format, finding it to be effective in duplicate detection. Rather than avoid and suppress duplicates, we focus on precisely finding and identifying clusters of similar pages to measure content reuse on the web (for commercial gain or otherwise) and detect phishing websites as part of an anti-phishing strategy.

Zhang et al. [48] build a near-duplicate detection technique by chunking documents at the sentence-level and matching their signatures across a variety of English and Chinese datasets (1.69–50.2×10^6 documents, 11–490 GB). We chunk documents at the paragraph-level, and compare the performance of matching at the file- and paragraph-level. We focus on applications in detecting commercialized duplication and web phishing, showing that our techniques can scale to web-size corpora (§5, 2.86×10^9 documents, 99 TB). Their signature creation also leverages prior work, using shingles [10], SpotSigs [44], and I-Match (SHA-1) [14], preferring I-Match and its efficiency. Our work validates SHA-1's performance, which we leverage to achieve precise and efficient detection.

Henzinger [24] compares the performance of algorithms that use shingling and Charikar's locality sensitive hashing (LSH). While LSH achieves better precision than shingling, combining the two provides even higher precision. Exploration of LSH is an interesting possible complement to our use of cryptographic hashing: although the objective of our paper is not a survey of algorithms, we briefly compare LSH and cryptographic hashing in §6.5.

Yang and Callan [46] 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. [26] develop an approximate matching algorithm for overlaps and content reuse detection in blogs and news articles. A search query is compared to sentence-level signatures for each document, with returned results being some Euclidean distance $\sqrt{d}$ of each other. Their system, tested on corpus sizes of 1-100×10^3 documents, balances the trade-off between a higher true positive rate (recall) with lower precision and their algorithm's quadratic runtime ($O(n^2)$). They also optimize part of their processing by using incremental updates for new content. We focus on precise matching in quasilinear time ($O(n \log n)$) on larger chunks (at the paragraph-level) of content reuse. In our applications, we look at detecting large-scale content reuse on web-scale corpora ($\geq 10^9$ documents), requiring high precision to avoid being overwhelmed with false positives (requiring costly post-processing).

Phish Detection: We summarize prior work here, from our previous, more detailed review [6]. Machine learning has been used to detect phish, by converting a website's content [23] or URL and domain properties [29] into a set of features to train on. Other approaches measure the similarity of phish and original sites by looking at their content and structure: similarities can be computed based on the website's visual features like textual content, styles, and layout [27]. Many of these approaches use approximate matching, which runs the risk of producing false positive detections, and machine learning techniques have high computational requirements (that would make them difficult to run on clients). Our use of precise content matching helps avoid false positives, runs efficiently in clients, and can provide a first pass that complements heavier approaches.

4 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.

4.1 Overview

Our general approach is to compute a hash for each data item, then use these 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 (§4.4.1).
(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 form a *chunk hash vector*[1] $H(\mathbf{c}_f)$.

Discovery: (§4.2)

(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*, §4.4.2)

Detection: (§4.3)

(4) *Simple Object Matching*: Given a labeled dataset of hashed chunks or files $\mathbf{L}$, find all matches $\in \mathbf{C}$ where its hash is $\in \mathbf{L}_o$. 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 *badness 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 $\text{contains}(\mathbf{L_c}, f)$ is greater than a threshold, we consider f to be a *partial target* in $\mathbf{T}$.
(6) *Bad Neighborhood Detection:* Apply stop-chunk removal (§4.4.2), then for each neighborhood $N = \{f_{N,1}, \ldots, f_{N,n}\}$ where the files share a hierarchical relationship, compute the overall *badness 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 $\text{badness}(N)$ is greater than a threshold, we consider N as a bad neighborhood in $\mathbf{T}_N$.

The thresholds for partial matching and bad neighborhood detection are configurable; we set the default threshold to one standard deviation over the mean. We elaborate on our choice and how to select a threshold in §4.2.

[1]While the vector contains an ordering of hashed chunks, we do not currently use the order.

4.2 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 (§8.1). One could also seed $\mathbf{L}$ with banking websites to identify phishing sites that reproduce this content (we seed $\mathbf{L}$ with PayPal in §8.2).

One can also identify $\mathbf{L}$ through a *blind discovery* process (Step 3b) that automatically finds widely duplicated content. Blind discovery is appropriate when an expert is unavailable, or if the source of copying is unknown. We first populate $\mathbf{L}_f$ 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 threshold to be higher when the dataset size is larger. We looked at the ROC curves (a plot between the true positive rate and false positive rate) 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 the Common Crawl dataset of $40.5{\times}10^9$ 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 $\mathbf{L}$: while common, they are not very significant or useful indicators of mass duplication. To make blind discovery more useful, we remove this very common but benign content using stop-chunk removal, described in §4.4.2.

Given a threshold, all chunks c 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 $\mathbf{L}$:

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

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. Parts of discovery are inherently parallelizable and allow for distributed processing by dividing the corpus to various workers; we use MapReduce to parallelize the work (§5). However, to maximize detection, the final data join and thresholding must consider the full corpus. 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 $\text{duplicates}(j) = \text{duplicates}(j_1) + \text{duplicates}(j_2)$: $j_1 \in \mathbf{C}_1$, $\text{duplicates}(j_1) = 1000$ and $j_2 \in \mathbf{C}_2$, $\text{duplicates}(j_2) = 100$. Object j only exceeds the threshold in the complete corpus (with $\text{duplicates}(j) = 1100$), not with consideration of only j_1 or j_2.

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 $O(n \log n)$.

4.3 Detection

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

In *simple object matching*, our targets **T** are an exact match of a chunk c or file f in **L**. Given **L**, find all chunks or files $\in$ **C** where its hash is $\in \mathbf{L}_o$ and add them to the set of targets **T**. We can then analyze **T** to understand if objects in **L** are being duplicated in **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 of where duplication happens.

Algorithmic performance of detection is $O(mn \log n)$, where m is the size of the labeled data, **L**, and n the size of the corpus **C**. Since $|L| << |C|$ (the corpus is large, with millions or billions of pages, only fraction of which are labeled as candidates of copy), performance is dominated by $O(n \log n)$ because of sorting. With optimized sorting algorithms (such as those in Hadoop), our approach scales to handle web-sized corpora.

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 **L**. Containment examines the chunk hash vector $H(\mathbf{c}_f)$ of each file to see what fraction of chunks are in **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 shallow hierarchies, so in the worst case each site is a neighborhood. For example, while people might easily create domains and spread content across them, the duplicated content would be detected as matches and reveal a cluster of neighborhoods (or sites) containing duplicated content. However, for complex sites with rich user content (e.g., GeoCities), 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 **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 (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

$$\text{detected}(\mathbf{L}, \mathbf{C}_1 \cup \mathbf{C}_2) = \text{detected}(\mathbf{L}, \mathbf{C}_1) \cup \text{detected}(\mathbf{L}, \mathbf{C}_2).$$

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 we showed earlier, 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 $O(n \log n)/p$.

4.4 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* and whitespace normalization. We evaluate the effectiveness of these methods in §6.1.

4.4.1 Detecting and Handling Recursion Errors. Crawling the real-world web is a perilous process, with malformed HTML, crawler traps, and other well understood problems [9, 25]. 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 §6.1, 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.

4.4.2 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> </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, and use both manual and automated methods of generating lists of stop chunks. We find that automated generation, while less accurate, is sufficient.

We manually generated lists of stop chunks for duplicate detection in the web (§7). In Common Crawl, the list of 226 chunks is short enough to allow manual comparison: if the list becomes too large, we can apply Bloom filters [8] to support efficient stop-chunk removal.

For automated generation, we apply the heuristic of treating all short chunks as stop chunks. In our evaluation of expert-identified content (§8), we discard chunks shorter than 100 characters. We also compare manual and automated generation: we previously labeled manually 226 (45%) of the top 500 chunks in Common Crawl as benign. By discarding chunks shorter than 100 characters, we automatically label 316 (63%) as benign: 222 benign and 94 non-benign from our manually labeled list. The non-benign that

are automatically labeled aren't necessarily "bad"—typically they are basic layout building blocks used in web templates or editors. Thus we find the trade-off acceptable: manual generation is more accurate, but automatic generation is sufficient for applications using expert-identified content.

4.5 Chunking and Hashing

Chunking text and data into non-overlapping segments has been done in natural language processing (NLP) [1, 37], in disk deduplication optimization [35, 47], and in information retrieval [36]. We chunk all content in our corpora, breaking at the HTML delimiters `<p>` (paragraph) and `<div>` (generic block-level component) tags, that are used to structure documents. We could also chunk on other tags for tables, frames, and inline frames, but find that our chosen delimiters are sufficiently effective.

Hash function: Unlike prior work with hashing from Natural Language Processing, we use *cryptographic* hashing to summarize data. We employ the SHA-1 [18, 32] cryptographic hash function for its precision—identical input always produces the same output, and different input yields a different output. Cryptographic hashing is used in disk deduplication [35], but most prior work considering duplicate detection uses locality-sensitive [15, 28] and semantic [38] hashes. We use cryptographic hashing to eliminate the small false positive rate seen in other schemes: we show in §6.5 that even a tiny rate of false positives is magnified by large corpora.

5 DATASETS AND IMPLEMENTATION

5.1 Datasets and Relevance

This paper uses three web datasets: Common Crawl, GeoCities, and our own phishing site corpus. We use the Common Crawl `crawl-002` dataset ($\mathbf{C}_{cc}$) collected in 2009/2010 and publicly provided by the Common Crawl Foundation [20] to represent recent web data. `crawl-002` is freely available on Amazon S3 and includes $2.86{\times}10^9$ items (26 TB compressed, 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 ($\mathbf{C}_g$) crawled by the Archive Team [4] just before the GeoCities service was shuttered by Yahoo! in October 2009. The dataset was compiled between April–October 2009 and contains around $33{\times}10^6$ files (650 GB compressed) including documents, images, MIDI files, etc. in various languages. Although the content is quite old, having been generated well before its compilation in 2009, it provides a relatively complete snapshot of diverse, user-generated content.

We generate the third dataset of phish ($\mathbf{C}_p$) by extracting the top-level webpages (HTML) from a stream of 2374 URLs of suspected phishing sites provided by PhishTank [33], a crowd-sourced anti-phishing service, over two days (2014-Sep-24 and 2014-Sep-25). (Our other datasets, Common Crawl and GeoCities, are not suitable for phish detection since phish lifetimes are often only hours to days, leaving very little time to crawl phish.) From the collected URL stream, we automatically crawl the suspect URLs and manually classify each as phish or otherwise. We ignore phishing sites that were removed by the time we crawl, discarding about 20% of the stream.

Dataset Relevance: As the web evolves quickly, its content and structure also evolves. Most GeoCities content dates from the late 1990s to the early 2000s, Common Crawl is from 2009–2010, and our phishing dataset is from 2014. Does evaluation of our techniques over these older datasets apply to the current web? We strongly believe it does, for two reasons: our datasets fulfill the requirement for content classification and we show that our approach easily adapts to today's and tomorrow's web.

First, the key requirement for classification of content in a corpus is that the corpus be diverse and large enough to approach real-world size and diversity. Both GeoCities and Common Crawl satisfy the diversity requirement. While GeoCities is perhaps small relative to the current web, we believe it is large enough provide diversity. Our phishing dataset is intentionally small because it addresses a more focused problem; it shows considerable diversity in that domain.

Second and more importantly, the web will always be different tomorrow, and continue to change over the next ten years. The increasingly dynamic and personalized nature of the web will modify the edges (like recommended links) but leave the core content unchanged and still detectable with hashing. We show that our approach works well over many years of web pages with only modest changes (for example, adding the use of `<div>` in addition to `<p>` to identify chunks). Our largest change was to shift from static web content to crawling a browser-parsed DOM in our phishing study (§8.2)—while conceptually straightforward, its implementation is quite different. This change allows us to accommodate dynamically-generated, JavaScript-only web content. We believe that this range of ages in our datasets strongly suggests that our approach (perhaps with similar modest changes) will generalize to future web practices, whatever they may be.

5.2 Implementation

We implement our methods and post-processing on cloud computing services (Amazon EC2) and a local cluster of 55 commodity PCs running Apache Hadoop [2]. Processing was done with custom MapReduce programs [16], Apache Pig [3], and GNU Parallel [43]. Our current cluster can intake data at a rate around 9 TB/hour.

We initially hash files and chunks in Common Crawl ($\mathbf{C}_{cc}$) on EC2 in $11.5{\times}10^3$ compute hours (18 real hours, ~$650), producing $2.86{\times}10^9$ file hashes and $40.5{\times}10^9$ chunk hashes along with backreferences to the original dataset (1.8 TB of metadata).

We use our local cluster to hash GeoCities ($\mathbf{C}_g$) in 413 compute hours (1.5 real hours) producing $33{\times}10^6$ file hashes and $184{\times}10^6$ chunk hashes along with backreferences (4.7 GB of metadata).

Overall performance is good as the corpus grows to the size of a large sample of the web. Although the theoretical bound on processing is the $O(n \log n)$ sort of hashes, in practice performance is dominated by scanning the data, an $O(n)$ process that parallelizes well with MapReduce. We see linear runtime on uncompressed dataset size ranges from 5 GB to 5 TB (we omit this graph due to space). We expect processing 225 TB of uncompressed data (Common Crawl, Feb. 2019, `CC-MAIN-2019-09`) on the same EC2 setup used earlier to take 41 real hours and $1300.

5.3 Reproducibility

Our research is reproducible, with all code available as open source (when possible), and all data available from us or provided with pointers to other public sources (for data generated by others).

Source code and data used for validation, discovering and detecting duplication of web content, and detecting clones of Wikipedia are available at https://ant.isi.edu/mega and https://github.com/cardi/content-reuse-detection. Source code for our AuntieTuna anti-phishing plugin and data used in our anti-phishing application are available at https://ant.isi.edu/software/antiphish and https://github.com/cardi/auntietuna. Instructions for reproducibility are included in their respective repositories.

6 VALIDATION

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

6.1 Do Our Cleaning Methods Work?

Initial analysis of our raw data is skewed by crawler errors, and identification of bad neighborhoods can be obscured by common benign content. We next show that our cleaning methods from §4.4 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 $913{\times}10^6$ 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 [5].

We next describe validation that our stop-chunk removal process (§4.4.2) 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 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 stop-chunk removal, we compare bad neighborhood detection with and without stop-chunk removal. Stop chunks dilute some pages and skews the quality of the badness ratio; if we do not remove stop-chunks in Common Crawl ($913{\times}10^6$ neighborhoods), we would detect $1.88{\times}10^6$ (2.35%) more "bad" neighborhoods than the $79.9{\times}10^6$ bad neighborhoods we find after stop-chunk removal. These additional $1.88{\times}10^6$ "bad" neighborhoods are false positives, mainly consisting of detected stop chunks, which would dilute the results above the detection threshold and reduce detection's precision: (true positives)/(true positives + false positives).

6.2 Can We Discover Known Files and Chunks?

We next turn to the correctness of our approach. We begin by validating and verifying 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 website (blog.archive.org as of 2013-Aug-22) containing roughly 4000 pages or files and duplicate the entire site 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 (C_{cc}), our blind discovery threshold is 10^3 (threshold is set using §4.2): all files that have more than 10^3 duplicates are labeled.

Figure 1 shows the results of this experiment in C_{cc} file frequency. This frequency-occurrence graph shows the number of occurrences (y-axis) that a file object has, given the amount of times it has been duplicated (x-axis). 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 entire blog moves it from unique, unduplicated content (a grey dot in the top left) to an outlying point with 4000 pages occurring 5000 times (as indicated by a labeled black triangle at $(x, y) = (5000, 4000)$). 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 parts 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 site had the entire site (4000 files) been duplicated different amounts of times (at $d = 100, 250, 1000$). When the injected site has been duplicated $\leq 10^3$ times (three blue circles on and to the left of the red threshold line), that site and corresponding files will not be automatically discovered; all points right of the red threshold line (the black triangle at $x = 5000$) will. Note that even with fewer duplications (blue circles left of the red threshold), the injected files duplicated fewer times will be visibly obvious outliers on the graph and may be detected with manual analysis or more sensitive automation (using additional analysis of the corpus or an iterative search to determine an optimal threshold as defined in §4.2).

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 by duplicating an entire website $d = 5000$ times, but add a different paragraph to the beginning and end of each 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 (mean: 48 chunks, median: 23, max: 428). Our discovery threshold is increased from 10^3 to 10^5, because the number of chunks in C_{cc} is much larger than the number of pages.

Figure 2 shows chunk-level evaluation of this scenario, with each dot representing a particular chunk. The red dotted line at $x = 10^5$ marks our discovery threshold: all 6000 chunks to the right this line are discovered, added to the labeled dataset, and further analyzed.

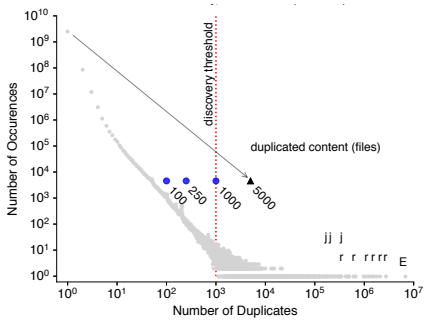

Figure 1: File-level discovery of injected duplicates (▲) in C_{cc}, compared to file frequency (grey dots). j: JavaScript, r: `robots.txt`, E: empty file.

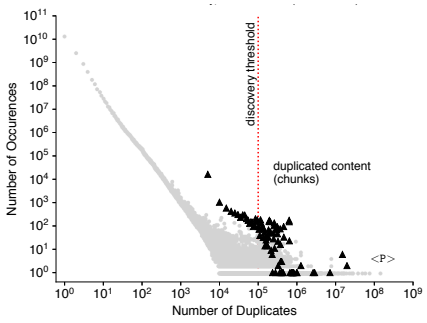

Figure 2: Chunk-level discovery of injected duplicates (▲) in C_{cc}, compared to chunk distribution (grey dots).

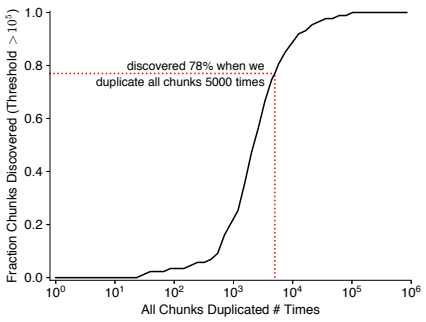

Figure 3: Percentage of chunks discovered in blog.archive.org given the number of times it is duplicated.

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 $1.2{\times}10^9$ chunks that make up the duplicated content of blog.archive.org (originally $242{\times}10^3$ chunks). The light grey dots 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.

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 the website $d = 5000$ 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.

6.3 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, as we increasingly mutate each page, at what point can we no longer detect it?

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 page

We build both $\mathbf{L}_{\text{expert}}$ and $\mathbf{L}_{\text{blind}}$ from $\mathbf{C}_i$ (as described in §6.2), 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 in $\mathbf{C}_i$ ($\overline{\text{cpp}} = 48$). 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 (Figure 4 in particular has no error bars).

This experiment shows that we can tolerate an additional 3.4× (using $\mathbf{L}_{\text{blind}}$) or 4.5× (using $\mathbf{L}_{\text{expert}}$) the mean number of chunks per page ($\overline{\text{cpp}}$) in each labeled dataset and still detect duplicated content. These tolerances are visually represented in Figure 4, where the blue ($\mathbf{L}_{\text{blind}}$) or green ($\mathbf{L}_{\text{expert}}$) dotted lines meet with the red dotted line (badness threshold): points to the left of the blue or green lines

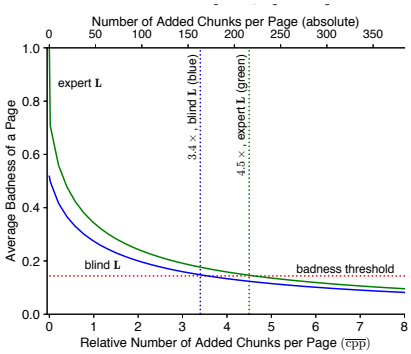

Figure 4: Effects of continuously adding chunks on pages.

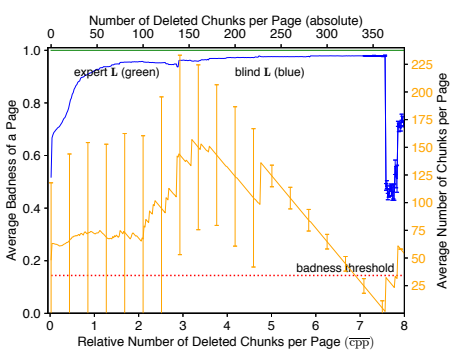

Figure 5: Effects of continuously deleting chunks on pages.

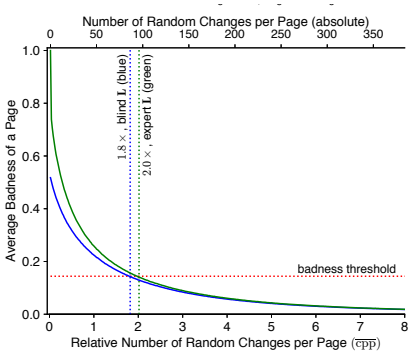

Figure 6: Effects of continuously changing chunks on pages.

on the x-axis will have an average badness above the detection threshold on the y-axis. This 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; deleting content will increase the badness ratio but may be overlooked because the number of total chunks on the page will be smaller. Users of hashing might require a minimum number of chunks per page before applying the badness ratio. 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, we initially see an increase in average badness of a page when using $\mathbf{L}_{\text{blind}}$ (blue) and stabilizes until a certain point as we increase the number of deleted chunks per page. Pages that have a small number of total chunks have on average a lower badness ratio until the page is eventually removed from the population, which results in a higher average badness as pages that have a higher number of total chunks survive deletion. In this experiment, our detection mechanism on average handles all 400 deletions (per page).

Similarly, we see a large variance in badness at the tail of the graph because the population of pages in $\mathbf{C}_i'^{(n)}$ (after mutation) decreases. As we increase the number of deleted chunks per page, the average number of chunks per page (orange) fall. Pages also cease to exist after all the chunks have been deleted; we see in Figure 5 that the average number of chunks per page increases as the population of pages decreases. 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 and change one random character in that chunk, 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.

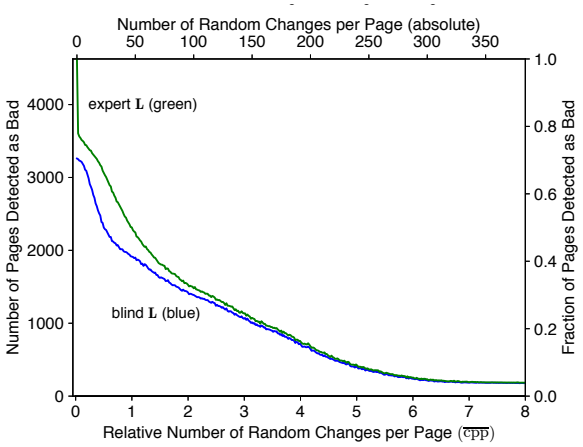

Figure 7: Number of pages detected as bad after continuously changing chunks on pages in blog.archive.org.

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 4600 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 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 ($|\mathbf{L}_{\text{expert}}| = 21{\times}10^3$, $|\mathbf{L}_{\text{blind}}| = 4.4{\times}10^3$). We next transition into the validation of detecting known bad neighborhoods.

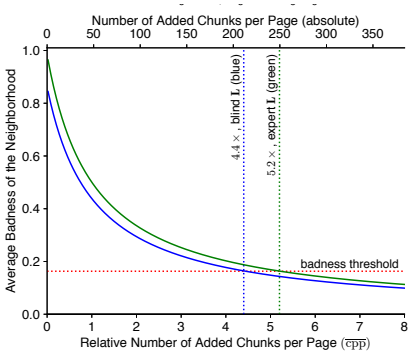

Figure 8: Effects of continuously adding chunks in a neighborhood.

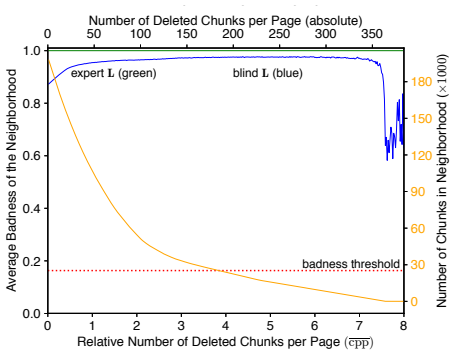

Figure 9: Effects of continuously deleting chunks in a neighborhood.

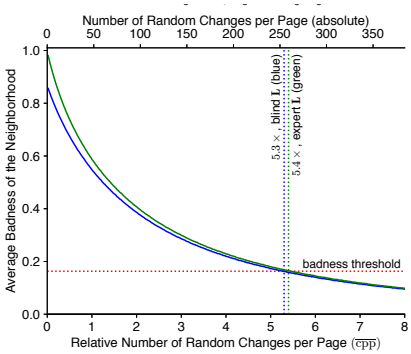

Figure 10: Effects of continuously changing chunks in a neighborhood.

6.4 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 (§6.3) 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 in N ($\overline{\text{cpp}}$) against the resulting badness ratio of the neighborhood after mutation ($N'^{(n)}$). We use a fixed badness threshold as described in §7.3. We again take the average over 10 runs over N at each x 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 use in two applications: content reuse detection over web content using the blind process and detection of expert-identified content in the web.

Table 1: Performance of expert-identified detection on phish corpus using cryptographic and locality-sensitive hashing.

True Nature of Page	Classified As	Hash Alg. Crypto	Hash Alg. LSH
PayPal Phish	PayPal Phish (TP)	43	43
	Missed PayPal (FN)	42	42
Non-PayPal	**Misclassfied PP Phish (FP)**	**0**	**10**
	Non-PayPal (TN)	1803	1793
Total		1888	1888

6.5 Cryptographic vs. Locality-Sensitive Hashes

Our work uses cryptographic hashing functions to minimize the impact of false positives that result from locality-sensitive and semantic hashing. To quantify this trade-off, we next compare bad page detection with SHA-1 (our approach) to the use of Nilsimsa [15], a locality-sensitive hashing algorithm focused on anti-spam detection. We use a corpus $\mathbf{C}_p$ of 2374 suspected phish (as described in §5) and build a labeled dataset $\mathbf{L}$ from current and recent PayPal U.S., U.K., and France home pages (Sep. 2014, plus Jan. 2012 to Aug. 2013 from archive.org).

We process the datasets, chunking on `<p>` and `<div>` tags, computing hashes of each chunk in $\mathbf{C}_p$ and $\mathbf{L}$ with both SHA-1 and Nilsimsa. We then use $\mathbf{L}$ to detect PayPal phish in $\mathbf{C}_p$. For detection with Nilsimsa, we use a matching threshold of 115 (0 being the fuzziest and 128 an exact match), a relatively conservative value.

Table 1 compares the confusion matrix when using SHA-1 (Crypto) and Nilsimsa (LSH) independently in detection. Both algorithms detect (TP = 43) and miss (FN = 42) the same number of PayPal phish. However, Nilsimsa has false positives, misclassifying 10 pages as PayPal phish, while SHA-1 misclassifies none.

Even very low, non-zero false-positive rates ($0<\text{FPR}<1\%$) are bad when used against web-size corpora, since false positives in a large corpus will overwhelm true positives. For Common Crawl with 2.86×10^9 files, Nilsimsa's very low 0.55% FPR at threshold 115 could result in 15.7×10^6 *false positives* (upper bound)!

Table 2: Categories of the top 100 distinct chunks in C_{cc}.

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

Table 3: Classification of a sample of 100 distinct chunks with more than 10^5 occurrences in C_{cc}.

Description	|c|	Type
Misc.	4	Benign
JavaScript	2	Benign
escaped	1	Benign
other	1	Benign
Templates	83	Benign
navigation	17	Benign
forms	32	Benign
social	4	Benign
other	30	Benign
Commercial	6	
spam	1	Malicious
JavaScript advertising	3	Ambiguous
JavaScript tracking	2	Ambiguous
Possibly Commercial	5	Ambiguous
Total	100	

LSH's design for approximate matching makes some false positives *inevitable*. A challenge with any LSH is finding the "right" threshold on each particular dataset to minimize the FPR. The number of false positives can differ greatly from small variations in threshold. We exhaustively studied the parameter space for Nilsimsa and our phishing dataset. A threshold of 128 forces exact matching, causing no false positives, but also makes the algorithm equivalent to cryptographic hashing. Our initial parameter choice was 115; all thresholds from 120 down to 115 give a very low but non-zero false-positive rate from 0.33% to 0.55%. At a threshold of 114, the false-positive rate doubles (1.11%) and as we continue to decrease the threshold, the FPR grows rapidly. Matching with thresholds from 128 to 120 is like exact matching with no false positives, in which case our analysis is needed to evaluate its performance. Although we find some thresholds with no false positives, in general, exhaustive search of the parameter space is not possible, and no one value will be "correct" across varying inputs.

The problem of false positives overwhelming rare targets known as the *base rate fallacy* and is a recognized barrier to the use of imprecise detection in security problems [7, 39]. This problem motivates our use of cryptographic hashing.

7 ANALYSIS OF BLIND DISCOVERY OF WEB COPYING

We next study the application of *blind discovery of duplication of web content*, and use this application to understand our approach.

7.1 Why is File-level Discovery Inadequate?

We first consider file-level discovery on both datasets. File-level comparisons are overly sensitive to small mutations; we use them to establish a baseline against which to evaluate chunk-level comparisons.

Figure 1 (grey dots) shows the long-tail distribution of the frequency of file-level hashes in Common Crawl (2.86×10^9 files, C_{cc}). We look at both the top 50 most occurring files and a sample of 40 random files that have more than 10^3 occurrences and find only benign content (e.g., JavaScript libraries, `robots.txt`). We see the same results with GeoCities (C_g), where common files include the GeoCities logo, colored bullets, and similar benign elements.

7.2 How Does Chunking Affect Discovery?

We expect the greater precision of chunk-level analysis to be more effective. We next consider chunking (§4.1) of textual files (HTML, plaintext, JavaScript) by paragraphs (i.e., the literal <p> tag).

Figure 2 shows frequency-occurrence distribution of the 40.5×10^9 chunks in Common Crawl (C_{cc}). Again, we see a heavy-tailed distribution: 40% of chunks are unique, but $\sim3.7\times10^9$ distinct chunks appear more than 10^5 times. The most common chunk is the empty paragraph (<p>).

Chunking's precision reveals several different kinds of duplication: *affiliate links*, JavaScript *ads*, *analytics*, and *scripts*, and *benign content* dominating the list. Table 2 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.

Again, we turn to a random sample of the tail of the graph 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 3.

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 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.

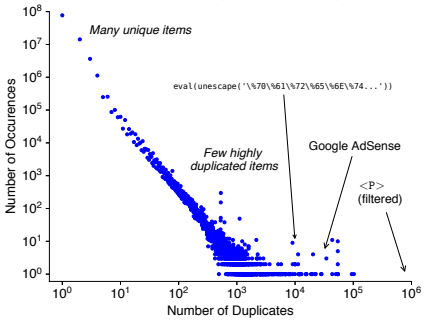

Figure 11: Chunk-level discovery on C_g ($97{\times}10^6$ chunks, after heuristic and stop-chunk removal).

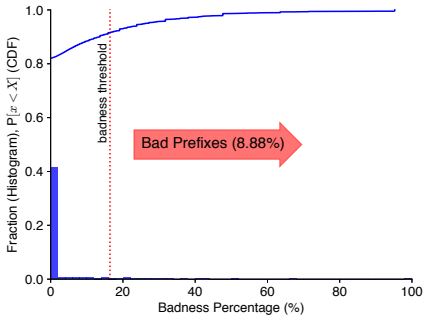

Figure 12: Frequency of badness of neighborhoods in C_{cc}, as a histogram (bars) and CDF (lines).

Description	$\|N\|$	%
Ads*	2	5.0
Blog*	19	47.5
Empty	1	2.5
Forms	1	2.5
Forum	1	2.5
"Suspect"	0	0.0
JavaScript	2	5.0
Templated/CMS	17	42.5
Total*	43	100

Table 4: Classification of a sample of 40 bad neighborhoods from C_{cc}. (*) indicates overlap between categories.

We confirm our results over a second dataset with chunk-level discovery on C_g (GeoCities) in Figure 11. We see a similar distribution overall, and find similar templates and JavaScript as in C_{cc}.

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

7.3 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 C_{cc} as our labeled dataset L_c (from §4.2), and identify bad neighborhoods in the full dataset using the algorithm in §4, step 6. C_{cc} contains $900{\times}10^6$ neighborhoods. Our detection threshold uses the mean and standard deviation across all neighborhoods.

As one would hope, most neighborhoods $N \in C_{cc}$ are not bad (91%). Figure 12 shows a combined histogram and CDF the bad content ratio of all neighborhoods. We observe that $79.8{\times}10^6$ prefixes (9%) out of $900{\times}10^6$ 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 $19.6{\times}10^6$ that are above the threshold and classify them in Table 4. 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 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 §8 with an expert-provided L.

In GeoCities: We repeat this experiment on GeoCities and obtain similar results. A random sample of 100 of the top 5% of worst neighborhoods found 40% to be link farms (clusters of ad-centric pages), while the other 60% were benign (usually templates). These results confirm what we find in Common Crawl. We believe the higher rate of copying in link farms reflects the greater susceptibility of search engines to duplicated content at this earlier time.

8 APPLICATIONS WITH EXPERT-IDENTIFIED CONTENT

We next look at two systems that use our approach, and use expert-identified content instead of blind discovery. Expert-identification is useful when targets locations are known but copied locations are unknown.

8.1 Detecting Clones of Wikipedia for Profit

We first explore finding copies of Wikipedia on the web. Although Wikipedia's license allows duplication [45], and we expect sharing across sibling sites (Wiktionary, Wikiquote, etc.), other copies are of little benefit for users and often serve mainly to generate advertisement revenue, support link farms, or dilute spam.

We next consider a copy of Wikipedia (from June 2008 [21], in English) as our labeled dataset L and use it to understand if Wikipedia is copied wholesale or just in parts.

Wikipedia becomes a L_c of $75.0{\times}10^6$ distinct chunks of length more than 100 characters (we treat shorter chunks as stop chunks, §4.4.2) and then search for this content in the Common Crawl corpus (C_{cc}, Nov. 2009 to Apr. 2010). Utilizing L_c, we identify bad neighborhoods in C_{cc} using the algorithm described in §4.

Table 5: Classification of the top 40 bad neighborhoods in C_{cc}, L = Wikipedia.

Description	$\lvert N \rvert$	%	Type
Wikipedia Clones/Rips	31	78	
"Wikipedia Ring"	13		Profit
Reference Sites	5		Profit
Ads	10		Profit
Fork	1		Ambig.
Unknown	2		Ambig.
Search Engine Optim.	3	8	
e-Commerce	2		Profit
Stock Pumping	1		Profit
Wikipedia/Wikimedia	5	13	Benign
Site using MediaWiki	1	3	Benign
Total	40	100	

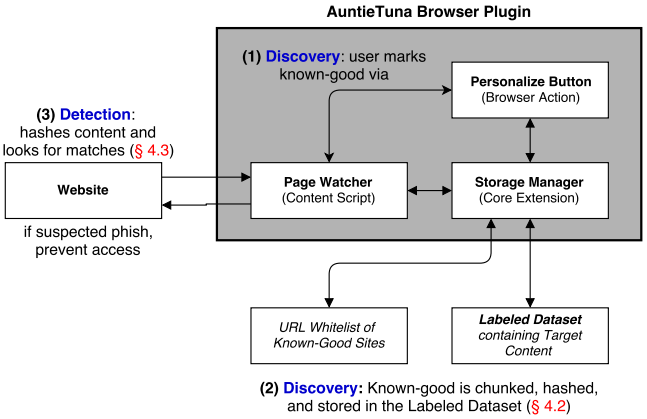

Figure 13: Implementation diagram of the AuntieTuna anti-phishing plugin.

The length of time between the crawl dates of **L** and C_{cc} may bias our detection's true positive rate in a particular direction. To understand Wikipedia's rate of change, during the 16–22 months between **L** and C_{cc}, Wikipedia added an additional 1.45–1.86×10^6 pages/month (an increase from 9.00×10^6 to 14.6×10^6 pages), encompassing 2.52–3.46 GB of edits/month [22]. Thus, if sites in C_{cc} copy from a more recent version of Wikipedia than **L**, we would expect that to bias our detection's true positive rate to be lower.

Our detection mechanism finds 136×10^3 target neighborhoods (almost 2% of 68.9×10^6 neighborhoods in C_{cc}) of path length 1 that include content chunks of length > 100 from Wikipedia. To understand how and why more than 100×10^3 sites copy parts of Wikipedia, we focus our analysis on neighborhoods that duplicate more than 1000 chunks from Wikipedia. We look at the 40 neighborhoods with the largest number of bad chunks and classify them in Table 5. We find 5 Wikimedia affiliates, including Wikipedia, Wikibooks, and Wikisource. More interestingly, we find 34 instances of duplicate content on third party sites: 31 sites rip Wikipedia wholesale, and the remaining 3 utilize content from Wikipedia subtly for search-engine optimization (SEO).

Almost all of the 31 third-party sites significantly copying Wikipedia are doing so 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 are similar, sometimes with the addition of other content. Finally, we also observe Wikipedia content used to augment stock pumping promotions or to draw visitors to online gambling.

Our study of Wikipedia suggests that our approach is very accurate, at least for bulk copies. All neighborhoods in our sample of the tail of the distribution were copies of Wikipedia, and only one site was a false positive (because it uses MediaWiki, an open-source wiki application). All others were true positives.

We have shown that from a labeled dataset (Wikipedia), our approach detects dozens of copies across the entire web, and that most of the bulk copies are for monetization. We next shift from bulk copying of Wikipedia to targeted copying in phishing sites.

8.2 Detecting Phishing Sites

Phishing websites attempt to trick users into giving up information (passwords or banking details) with replicas of legitimate sites—these sites often duplicate content, making them detectable with our methods. We adapt our system with an expert-labeled dataset built from common targets (such as real banking pages). In our prototype browser extension, AuntieTuna (implementation diagram in Figure 13, described in our prior work [6]), users first identify pages they care about (manually or automatically, via Trust on First Use) to build a custom labeled dataset (Discovery, §4.2). AuntieTuna then checks each page the user visits for potential phish (Detection, §4.3). As an alternative system implementation, phish could be detected centrally by crawling URLs found in email or website spam, then testing each as potential phish with our method.

To evaluate our approach to detect phish, we examine PayPal phishing and build a labeled dataset $\mathbf{L}_{pp}$ with PayPal home pages, chunking on both `<p>` and `<div>`, resulting in 311 distinct chunks longer than 25 characters. Our corpus C_p is drawn from a stream of 2374 suspected phish from over 2 days of data (Oct. 2014) from PhishTank [33], a crowd-sourced anti-phishing site. All operations are done on the DOM content (as rendered by the browser). We compare each suspected phish against $\mathbf{L}_{pp}$ using our algorithm (§4), which we've shown to be robust to minor changes (§6) with a detection threshold of one or more chunks.

To evaluate ground truth, we manually examine C_p and identify 85 phish using text content from PayPal. Our mechanism detects 50 (58.8%) pages: 43 directly copy from the original (whitespace normalization included), and an additional 7 obfuscate their copies with JavaScript. Table 6 classifies the type of techniques each phish uses. Our targeted dataset and precise approach prevents false positives (specificity is 100%), although we see 40% of phish copy too little content from the original for us to detect (in this study we ignore image-based phish). This evaluation shows that our hash-based detection can be a part of an anti-phishing scheme, complementing other techniques.

Table 6: Classification of phish in C_p, L_{pp} = PayPal.

Description	Num. Pages	%	
Candidates	2374		
Unavailable	486		
Ripped	1888		
Other	1764		TN = 1764
PayPal (image-based, *removed*)	39		
PayPal	85	100.0	FP = 0
Successfully detected	50	58.8	TP = 50
Direct copies	35		
Whitespace normalization	8		
JavaScript obfuscation	7		
Custom-styled with minor PayPal content	35	41.2	FN = 35

9 CONCLUSION

In this paper, we developed a method to *discover* previously unknown duplicated content and to *precisely 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, webpage spam, and phishing websites.

ACKNOWLEDGMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-18-2-0280, and the Defense Advanced Research Projects Agency (DARPA) under contract W911NF-18-C-0020. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of DARPA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] Steven Abney. 1991. Parsing by Chunks. Principle–based Parsing.
[2] Apache. [n. d.]. Hadoop. http://hadoop.apache.org.
[3] Apache. [n. d.]. Pig. http://pig.apache.org.
[4] ArchiveTeam. 2009. GeoCities. http://archiveteam.org/index.php/GeoCities.
[5] Calvin Ardi and John Heidemann. 2014. *Web-scale Content Reuse Detection (extended).* Technical Report ISI-TR-692. USC/ISI.
[6] Calvin Ardi and John Heidemann. 2016. AuntieTuna: Personalized Content-based Phishing Detection. In *Proceedings of the NDSS Workshop on Usable Security (USEC '16).* https://doi.org/10.14722/usec.2016.23012
[7] Stefan Axelsson. 2000. The Base-rate Fallacy and the Difficulty of Intrusion Detection. *ACM Trans. Inf. Syst. Secur.* 3, 3 (Aug. 2000), 186–205. https://doi.org/10.1145/357830.357849
[8] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692
[9] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International World Wide Web Conference.* 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X
[10] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. 1997. Syntactic clustering of the Web. In *Selected papers from the sixth international conference on World Wide Web.* 1157–1166. http://dl.acm.org/citation.cfm?id=283554.283370
[11] Moses S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing (STOC '02).* ACM, New York, NY, USA, 380–388. https://doi.org/10.1145/509907.509965
[12] Stanford Chiu, Ibrahim Uysal, and W. Bruce Croft. 2010. Evaluating text reuse discovery on the web. In *Proceedings of the third symposium on Information interaction in context (IIiX '10).* 299–304. https://doi.org/10.1145/1840784.1840829
[13] Junghoo Cho, Narayanan Shivakumar, and Hector Garcia-Molina. 2000. Finding Replicated Web Collections. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00).* ACM, New York, NY, USA, 355–366. https://doi.org/10.1145/342009.335429
[14] Abdur Chowdhury, Ophir Frieder, David Grossman, and Mary Catherine McCabe. 2002. Collection Statistics for Fast Duplicate Document Detection. *ACM Trans. Inf. Syst.* 20, 2 (April 2002), 171–191. https://doi.org/10.1145/506309.506311
[15] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. 2004. An Open Digest-based Technique for Spam Detection. *ISCA PDCS* 2004 (2004), 559–564. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.6185
[16] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04).* USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264
[17] Heinz Dreher. 2007. Automatic Conceptual Analysis for Plagiarism Detection. *Journal of Issues in Informing Science and Information Technology* 4 (2007), 601–614. https://doi.org/10.28945/3141
[18] D. Eastlake 3rd and P. Jones. 2001. *US Secure Hash Algorithm 1 (SHA1).* Technical Report 3174. http://www.ietf.org/rfc/rfc3174.txt Updated by RFCs 4634, 6234.
[19] Sven Meyer Zu Eissen and Benno Stein. 2006. Intrinsic Plagiarism Detection. In *Proceedings of the European Conference on Information Retrieval ECIR-06.*
[20] Common Crawl Foundation. [n. d.]. Common Crawl. http://commoncrawl.org.
[21] Wikimedia Foundation. 2008. Static HTML Dump of Wikipedia. http://dumps.wikimedia.org/other/static_html_dumps/2008-06/en/
[22] Wikimedia Foundation. 2019. Wikimedia Statistics. https://stats.wikimedia.org/v2/#/en.wikipedia.org [Online; accessed 2019-Feb-27].
[23] R. Gowtham and Ilango Krishnamurthi. 2014. PhishTackle–a Web Services Architecture for Anti-phishing. *Cluster Computing* 17, 3 (Sept. 2014), 1051–1068. https://doi.org/10.1007/s10586-013-0320-5
[24] Monika Henzinger. 2006. 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).* ACM, 284–291. https://doi.org/10.1145/1148170.1148222
[25] Allan Heydon and Marc Najork. 1999. Mercator: A Scalable, Extensible Web Crawler. *World-Wide Web Journal* 2, 4 (Dec. 1999), 219–229. https://doi.org/10.1023/A:1019213109274
[26] Jong Wook Kim, K. Selçuk Candan, and Junichi Tatemura. 2009. 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).* ACM, 81–90. https://doi.org/10.1145/1526709.1526721
[27] Wenyin Liu, Xiaotie Deng, Guanglin Huang, and Anthony Y Fu. 2006. An antiphishing strategy based on visual similarity assessment. *Internet Computing, IEEE* 10, 2 (2006), 58–65. https://doi.org/10.1109/MIC.2006.23
[28] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *Proceedings of the International Conference on Very Large Data Bases.* VLDB Endowment, Vienna, Austria, 950–961. http://dl.acm.org/citation.cfm?id=1325958
[29] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. 2009. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09).* ACM, 1245–1254. https://doi.org/10.1145/1557019.1557153
[30] J. Malhotra and J. Bakal. 2015. A survey and comparative study of data deduplication techniques. In *2015 International Conference on Pervasive Computing (ICPC).* 1–5. https://doi.org/10.1109/PERVASIVE.2015.7087116
[31] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web (WWW '07).* ACM, 141–150. https://doi.org/10.1145/1242572.1242592
[32] National Institute of Standards and Technology. 2008. *Secure Hash Standard (SHS).* Federal Information Processing Standard (FIPS) 180-3. National Institute of Science and Technology. http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
[33] OpenDNS. [n. d.]. PhishTank. http://www.phishtank.com
[34] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. 2007. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation (NSDI'07).* USENIX Association, Berkeley, CA, USA, 2–2. https://www.usenix.org/legacy/event/nsdi07/tech/full_papers/pucha/pucha.pdf

[35] Sean Quinlan and Sean Dorward. 2002. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*. USENIX Association, Berkeley, CA, USA, Article 7. http://dl.acm.org/citation.cfm?id=1083323.1083333

[36] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. 2004. Automatic Detection of Fragments in Dynamically Generated Web Pages. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, New York, NY, USA, 443–454. https://doi.org/10.1145/988672.988732

[37] Lance A. Ramshaw and Mitchell P. Marcus. 1995. Text Chunking using Transformation-Based Learning. *ACL Third Workshop on Very Large Corpora* cmp-lg/9505040 (1995), 82–94. http://arxiv.org/abs/cmp-lg/9505040

[38] Ruslan Salakhutdinov and Geoffrey Hinton. 2009. Semantic hashing. *Int. J. Approx. Reasoning* 50, 7 (July 2009), 969–978. https://doi.org/10.1016/j.ijar.2008.11.006

[39] Bruce Schneier. 2005. Why Data Mining Won't Stop Terror. *Wired Magazine* (9 March 2005). https://schneier.com/essays/archives/2005/03/why_data_mining_wont.html

[40] Antonio Si, Hong Va Leong, and Rynson W. H. Lau. 1997. CHECK: a document plagiarism detection system. In *Proceedings of the 1997 ACM symposium on Applied computing (SAC '97)*. ACM, 70–77. https://doi.org/10.1145/331697.335176

[41] Neil T. Spring and David Wetherall. 2000. A Protocol-independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*. ACM, New York, NY, USA, 87–95. https://doi.org/10.1145/347059.347408

[42] Benno Stein, Moshe Koppel, and Efstathios Stamatatos. 2007. Plagiarism Analysis, Authorship Identification, and Near-duplicate Detection PAN'07. , 4 pages. https://doi.org/10.1145/1328964.1328976

[43] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. http://www.gnu.org/s/parallel

[44] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. 2008. 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)*. ACM, 563–570. https://doi.org/10.1145/1390334.1390431

[45] Wikipedia. 2017. Reusing Wikipedia Content. https://en.wikipedia.org/wiki/Wikipedia:Reusing_Wikipedia_content [Online; accessed 23-Feb-2019].

[46] Hui Yang and Jamie Callan. 2006. 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)*. ACM, 421–428. https://doi.org/10.1145/1148170.1148243

[47] Lawrence You, Kristal Pollack, and Darrell D. E. Long. 2005. Deep Store: An Archival Storage System Architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. https://doi.org/10.1109/ICDE.2005.47

[48] Qi Zhang, Yue Zhang, Haomin Yu, and Xuanjing Huang. 2010. Efficient Partial-duplicate Detection Based on Sequence Matching. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '10)*. ACM, New York, NY, USA, 675–682. https://doi.org/10.1145/1835449.1835562