Comparison of Texts Streams in the Presence of Mild Adversaries

Michael Malkin
mikeym@stanford.edu
Gates Building, room 464
Stanford, CA 94305

Ramarathnam Venkatesan
venkie@microsoft.com
One Microsoft Way
Redmond, WA, 98052

Abstract

Text sifting is a method of quickly and securely identifying documents for database searching, copy detection, duplicate email detection and plagiarism detection. A small amount of text is extracted from a document using hash functions and is used as the document's fingerprint. We build upon previous work by Broder et al. [4,5] and Heintze [8], specifically addressing a certain set of attacks that we discovered to be very powerful against previous systems. We achieve robustness against these attacks with a new selection process. We also give theoretical and experimental results for these and other attacks on text sifting functions.

Keywords: Text sifting, text recognition, plagiarism, document fingerprint, shingling, min-hash

1 Introduction

This paper presents an attack-resistant method for identifying plagiarized documents called *text sifting*. Text sifting systems form document fingerprints by extracting a small amount of text from each document, and then use these fingerprints to compare documents. We build upon the work of Broder et al. [4,5] and Heintze [8] (see Section 1.2).

The security of text sifting systems systems comes from the unpredictability of the document fingerprints. The idea is that an attacker without the secret key cannot know what a document's fingerprint will be, so they can only attack the document by making many changes and hoping that the changes significantly alter the document's fingerprint. It turns out that systems which use a sliding window to generate fingerprints are susceptible to powerful attacks. One of our contributions is to consider general attacks on such systems, as well as specific attacks on the sliding window used in previous systems. We improve security with a more robust selection process that replaces the sliding window.

The purpose of a text sifting system is to detect when documents are similar. We assume that there is an intelligent adversary, a plagiarist, who knows that a system will be used to detect similar documents and takes steps to circumvent it. The adversary would like to do so by making as few changes to documents as possible. In reality, of course, there may not be an intelligent adversary; the "adversary" may be human error or chance.

A text sifting function takes a document and produces a fingerprint which consists of *clusters*. Clusters are groups of words from the document which are ordered as in the document but are not necessarily contiguous in the document. Text sifting fingerprints have a few important properties:

1. *Random-looking:* The fingerprint is calculated deterministically using a secret key, but the clusters which are output can not be predicted without the key.
2. *Self-synchronizing:* If a cluster appears in two documents and is chosen for the fingerprint of one document, it has a good chance of being chosen for the fingerprint of the other document.
3. *Attack resistant:* The output should remain substantially unchanged when the original document is subject to attacks.

Previous systems (see Section 1.2) have implemented the first and second properties, but remain susceptible to certain attacks. For example, if a system used a sliding window of length 10, an attacker could change a document so that every ten-word span contained a change. The system would report that the attacked document was completely different from the original document. Text sifting is designed to identify documents in spite of this sort of attack.

1.1 Applications

Plagiarism and Copy Detection: The obvious application of text sifting is detecting plagiarism and copyright infringements. This not only means finding exact duplicates of documents, it also means finding close matches that have been changed to avoid copy detection mechanisms. Instead of combing through an entire database to see if a document has been plagiarized, the database might contain a text sifting fingerprint alongside each document. Searching for matches in the fingerprints is much faster than searching through the documents, and the attack-resistance of text sifting means that it is hard for an adversary to "disguise" documents. If a match is found among the fingerprints, further comparisons can be done on the full documents to avoid false matches.

An adversary can mount a plagiarism attack if it has access to a very large number of document fingerprints (see Section 3). Therefore, in copy detection systems which employ text sifting, the document fingerprints must be kept secret. Allowing unfettered public access to a copy detection system is also a security risk, as an attacker can iteratively modify a document to find the minimum modification necessary to produce a false negative. Heintze [8] discusses methods for allowing a text sifting system to be used by the public.

Duplicate Email Detection: Another application of text sifting is duplicate email detection. Email providers detect spam in part by counting the number of times the same email arrives at their servers. Spammers get around this by changing their email every time it is sent. It would be infeasible for an email service to fully compare every email that is received for every user to every other email that has

been recently received, but text sifting can be used to speed up the process and notice the similarities between emails even after attacks by spammers.

Duplicate email detection is essentially the same as plagiarism detection, but optimized for email. Instead of operating on words as is done in plagiarism detection, operations are on letters. Additionally, the preprocessor is much more aggressive, to take into account the tricks that spammers use. For example, the string "|<" (pipe – less-than) could be converted to the letter "K", while the letter "I" and the number "1" could be collapsed to the same token.

Note that a spammer is implicitly trying to perform a plagiarism attack by copying their own email and changing it enough that an email provider will not recognize it.

1.2 Previous Work

There are several companies who offer the service of detecting plagiarism on the web, for example Glatt Plagiarism Services [1] and Plagiarism.org [2], but their services are proprietary and they do not reveal their methodology.

The field of natural language processing is devoted to analyzing text by understanding its meaning. See, for example, Manning and Schütze [10].

Stanford Digital Library's SCAM project by Shivakumar and Garcia-Molina [14, 15, 16] breaks documents into non-overlapping chunks, stores all of these chunks in a database, and analyzes documents based on the frequency of the chunks it contains. Storage size per document is relatively large.

Heintze [8] considers plagiarism detection. One suggested method is to sort the hash values of groups of characters and use the groups with the smallest hash values as a document fingerprint. They conclude, however, that this method produces too large a probability of false-positive matches, and so use letter frequency to decide which groups of characters are used in document fingerprints.

Broder et al. [4, 5] suggest two methods for selecting groups of words. The first method is to use min-hashing, the second is to select groups of words equal to 0 modulo a constant. These are the methods used in our sifting stage (see Section 2.3), but our cluster formation stage avoids the use of sliding windows (see Section 2.2). The focus of their work is on duplicate document detection in order to find groups of similar documents on the web, not plagiarism detection.

Schleimer et al. [13] present the idea of winnowing. They use a sliding window to generate local document fingerprints which can guarantee that short substring matches between documents are ignored, while long substring matches are always found. They also consider plagiarism detection.

These last three works are similar to the text sifting work described in this paper. We build most upon the work of Broder et al. [4, 5], but where they use a sliding window to select text for document fingerprints, we present a more robust selection process.

2 Text Sifting

Text sifting consists of three phases. In practice these phases can be combined, but the algorithm is more easily explained as a three-pass process.

1. **Preprocessing:** The document is put into canonical form.
2. **Cluster Formation:** The canonical form is converted into a list of clusters.
3. **Sifting:** A subset of the clusters are chosen and output.

The first phase is common to most plagiarism detection systems. The second phase is unique to text sifting. The last phase is the same as the system presented in Broder et al. [4, 5].

Text sifting is slower than sliding window systems for two reasons. First, there is the overhead of the cluster formation stage, which is not present in sliding window systems. Second, when using a sliding window, an incremental hash function can be used in the sifting stage. This is not possible with text sifting, so a slower, non-incremental hash function must be used.

We use the generic term *token*, which is the minimum text element upon which a text sifting program operates. Tokens can be words, letters, or groups of several letters, allowing flexibility in implementation. For plagiarism detection, tokens would be words. For duplicate email detection, tokens would be letters.

A *cluster* is list of tokens ordered as they appear in the document. Similar structures are referred to as *shingles* by Broder et al. [4, 5], *substrings* by Heintze [8], and *k-grams* by Schleimer et al. [13]. These three structures are composed of contiguous tokens, while clusters are composed of tokens which are not necessarily contiguous in the document. Table 1 shows an example.

Many secure, universal hash functions are called for in this section. All such functions are based on the secret key, K. In general, any secure, universal hash functions will be sufficient, although faster hash functions are always preferred over slower hash functions. Universal hash functions are discussed by Cormen et al. [7]. Secure cryptographic hash functions are discussed, for example, by Stinson [17]. Broder et al. [5] use Rabin fingerprints [12].

2.1 Preprocessing Stage

The purpose of the preprocessing stage is to remove superfluous details that do not distinguish one document from another. First, filetype details are removed, for example by extracting the body text from PDF, Postscript, or HTML files.

Next, documents are put into canonical form. The specific form depends on the application. A plagiarism detection program would remove all characters that are not letters, numbers, or spaces, convert all white space into single spaces, and convert all letters to lowercase. A spam detection program might go further, removing all punctuation and white space and converting all characters and pairs of characters into canonical representations. For example, "I" and "1" could be converted to the same token, as could "K" and "|<".

At this point "common" words may be removed from the document. In plagiarism detection, this would include words like "a", "the", etc.

Next, all characters are converted into numbers. For plagiarism detection, this would mean assigning each token a value from 0 to 35, with 0 to 9 corresponding to the digits and 10-35 corresponding to the letters a to z. A permutation on all the possible values (0 – 35 in this case) is chosen pseudo-randomly based on the secret key, and all letters are permuted.

Finally, *token hashes* are generated by hashing the tokens with a secure, universal hash function. The output of the preprocessing stage is a list of the tokens hashes.

The document "quick brown fox jumps" contains four tokens: "quick", "brown", "fox", and "jumps". The following are all possible clusters from this short document:

1-token clusters	2-token clusters	3-token clusters	4-token clusters
(quick)	(quick brown)	(quick brown fox)	(quick brown fox jumps)
(brown)	(quick fox)	(quick brown jumps)	
(fox)	(quick jumps)	(quick fox jumps)	
(jumps)	(brown fox)	(brown fox jumps)	
	(brown jumps)		
	(fox jumps)		

Table 1: All possible clusters are shown for a very short document.

2.2 Cluster Formation Stage

The cluster formation stage is what distinguishes text sifting from other methods. The common method [4, 5, 8, 13] is to generate clusters (or their equivalent) with a sliding window, taking all blocks of contiguous tokens. Text sifting generates clusters of tokens which are not necessarily contiguous. The cluster formation stage takes as input a list of token hashes and outputs clusters of these token hashes.

We present two methods which form clusters while ignoring certain tokens: *random skipping* and *random partitioning*. They can be used independently or together. If random partitioning is used without random skipping, then it is used with a sliding window.

Random Skipping: The basic idea is to keep a cumulative hash of all the tokens in a cluster, and add new tokens based on what would happen to the cumulative hash. If adding a given token would put the cumulative hash in the *add* category, then add the token to the cluster. If adding the token would put the cumulative hash in the *stop* category, then add the token to the cluster and stop adding more tokens. Otherwise, skip the token under consideration and move on to the next token in the list.

In more detail, let all token hashes be in the set G. Let $g(\cdot,\cdot)$ be a function that takes as input two elements of G and outputs a single element of G. g should have the property that if a is known and b is chosen uniformly at random, or vice-versa, then $g(a,b)$ is unpredictable. For example, $g(a,b)$ could be equal to $a \oplus b$ or $a+b \bmod p$, depending on G.

Let $h(\cdot)$ be a secure, universal hash function that takes as input an element of G and outputs an element in the set H. Let A (meaning *add*) and S (meaning *stop*) be subsets of H such that A and S are disjoint.

Each token hash is assigned to start a single cluster. When a cluster is started, it contains only the single token hash that started it. It is assigned a cumulative hash $c \in G$ equal this token hash.

Now consider a partially-formed cluster of 1 or more token hashes. Let c be the cumulative hash for this cluster. Let t be the next token hash being considered for inclusion in the cluster. If $h(g(c,t)) \in S$ then add t to the cluster and stop adding tokens. If $h(g(c,t)) \in A$ then add t to the cluster and continue to the next token hash. Otherwise, just continue to the next token hash.

A and S may vary with the number of token values already in a cluster. For example, setting $|S| = 0$ when choosing tokens $2, 3, ..., l-1$ and setting $|A| = 0$ when choosing token l ensures that all clusters will consist of exactly l tokens. We have found experimentally that constant-length clusters are superior to variable-length clusters at detecting plagiarism.

The size of $|A|+|S|$ relative to $|H|$ determines how many token hashes will tend to be skipped between accepted token hashes when forming a cluster. On average, $\frac{|H|}{|A|+|S|} - 1$ token hashes will be skipped before one is accepted. For example, when $|H| = 2(|A|+|S|)$, half of all token hashes are skipped, and there is an average of one token hash skipped between every two accepted token hashes.

Random Partitioning: In random partitioning, the cluster formation algorithm is run multiple times and each time a subset of token values is ignored. First, the token hash values are partitioned into b sets of approximately equal size using the secret key K. Next, the cluster formation algorithm is run b times, each time running as normal but ignoring all token hashes in one of the b sets. If an attacker tries to change a document by adding, for example, the word, "very" in many different positions, at least 1 of the b iterations will ignore the word "very", increasing the chance of recognizing the attack. We have found experimentally that $b = 2$ is optimal.

Random Skipping with Random Partitioning: To combine random skipping with random partitioning, perform random partitioning, and for each of the b iterations of the cluster formation algorithm, run random skipping. In practice this is done in parallel, so the document is only scanned a single time.

2.3 Sifting Stage

This stage takes as input the clusters from the cluster formation stage, hashes them to find the *cluster hashes*, and outputs a small subset of the cluster hashes. Although cluster formation and sifting are explained as two separate processes, in practice each cluster is sifted as it is generated.

There are two methods of sifting, *pure hashing* and *min-hashing*. Both are based on the work of Broder et al. [4, 5]. Both methods use $H_K(\cdot)$, a secure, universal hash function with key K, which takes as input a series of token hashes and outputs a single cluster hash. The output of H_K must be large enough that it is very unlikely that two distinct clusters will have the same cluster hashes.

Pure Hashing: For each cluster c_i, $H_K(c_i)$ is output if $H_K(c_i) = 0 \bmod s$ for some fixed s. Afterwards, the cluster hashes are sorted to remove duplicates and speed up later comparisons.

Since each cluster is considered independently of all other clusters, a change in one cluster will not affect whether or not another cluster is output. s may be constant, or it may change with the length of the document being sifted. If s is constant, then the output will have length proportional to the length of the input document. To limit the size of the output, Broder [4] suggests using $s = 2^j$ and using larger j for larger documents.

The size of the output is nondeterministic. For a random input of n clusters with no duplicate clusters, and a constant s for all documents, the output size has a mean of $\frac{n}{s}$ and a standard deviation of $\sqrt{\frac{n}{s} \cdot \frac{s-1}{s}}$.

Min-Hashing: Hash all clusters with $H_K(\cdot)$ and sort them according to hash value. Remove duplicate hash values and output, in sorted order, the m smallest cluster hashes. If there are fewer than m unique cluster hashes then output all unique cluster hashes.

m can be constant or can depend on the length of the document. If m is constant, it will be difficult to detect a small document that has been embedded inside a larger document; since cluster hash values are distributed uniformly, there is a good chance that the larger document will have many clusters with smaller hash values than those in the smaller document, so many of the clusters which are in the smaller document will be pushed off the end of the sorted list. Allowing m to grow with document size increases the number of clusters in the small document that remain on the sorted list for the large document. The downside of this, of course, is increased storage space to store the extra clusters.

Changing one part of a document can affect the output of a cluster in another part of the document. For example, if cluster x has the m^{th} smallest hash value in a document, and a cluster y is added to the document such that the $H_k(y) < H_k(x)$, then x will be removed from the output and y will be added. This will occur even if x and y are far apart in the document.

Output Size: The optimal output size is specific to the application. It will be different for a full-web search than for a database of class essays, for example. Since attacks are detected randomly, increasing the output size will help to detect attacks. With pure hashing, the output size is either approximately linear in the size of the input or approximately constant. With min-hashing, the output size can vary arbitrarily with the size of the input, although constant, linear, or sublinear (for example, square root) relationships are best.

A constant-length output performs best when comparing documents of similar sizes. When comparing documents of different sizes, a constant output size must be carefully chosen. If the output is too large, space will be wasted when sifting small files. If the output is too small, fingerprints for large documents will not be very representative of the document.

When documents of radically different sizes are to be compared, a linear output size can be useful, especially in plagiarism detection. A common plagiarism attack is to insert a small file into a large file. If the output size is constant, the fingerprint of the large file with the plagiarism might not have many matches with the original, small file. A linear output size will make it more likely that this sort of attack will be detected. The disadvantage of using linear-size outputs is that the total text sifting overhead will tend to be much larger than for systems with constant output size.

2.4 Comparing Documents

In a text sifting system, documents are compared by comparing their fingerprints. The basis of comparison is the number of cluster hashes that the two fingerprints have in common. However, this number needs to be normalized to convey much meaning.

The most obvious normalization is to divide the number of shared cluster hashes with the total number of cluster hashes in both documents. If C_1 is the set of cluster hashes from document 1, and C_2 is the set of cluster hashes from document 2, then the first similarity measure is

$$S_1(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

This measure intuitively captures the idea that two documents are similar if they share many clusters in common, but does not do well in the situation where one fingerprint is much larger than the other. For example, it is possible to have a small S_1 even though $C_1 \subset C_2$, if $|C_2| \gg |C_1|$.

A similarity measure which avoids this problem is

$$S_2(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1|}$$

Broder [4] refers to this as an estimate of the *containment* of document 1 in document 2 when using pure hashing. This follows the intuition that document 1 is contained in document 2 if a large fraction of the clusters in document 1 appear in document 2. Note that $S_2(C_1, C_2)$ and $S_2(C_2, C_1)$ are not necessarily equal.

Finally, we present a third similarity measure, which is the maximum of $S_2(C_1, C_2)$ and $S_2(C_2, C_1)$.

$$S_3(C_1, C_2) = max\left(\frac{|C_1 \cap C_2|}{|C_1|}, \frac{|C_1 \cap C_2|}{|C_2|}\right)$$

If the fingerprint of either document is a subset of the other document's fingerprint, then S_3 will be equal to 1. If S_1 is large then S_3 will also be large, but S_3 may be large even when S_1 is small.

$$S_3(C_1, C_2) \geq S_1(C_1, C_2)$$

We prefer S_3 as a similarity measure for plagiarism detection.

See Broder [4] for more on computing document similarity.

3 Security of Text Sifting

Broder et al. [5] and Heintze [8] have demonstrated that these methods work well in the absence of adversaries. We now consider the robustness of text sifting functions against attacks. First we consider the theoretical performance of text sifting against several categories of attacks, and then describe the experimental results of a some of these attacks.

Certain attacks are beyond the scope of text sifting systems. One such class of attacks are attacks which do not leave documents in human-readable form, so for example, compression and encryption are not considered. Attacks which do not produce text files are also not considered, so we ignore, for example, attacks which convert documents into images. Finally, attacks which change all the words in a document are ignored. This could include translating a document to another language, or writing a new document that conveys similar information to the original document.

Text sifting is not necessarily secure if an adversary has direct access to a very large number of document fingerprints. Such an adversary could conceivably discover a small set of clusters which are 0 mod s (pure hashing) or which have small hash values (min-hashing) and insert those clusters into documents. This would not necessarily destroy the security of a text sifting system, but could decrease its effectiveness. Therefore, it is assumed that adversaries do not have direct access to document fingerprints. Of course, an adversary with knowledge of the secret key can mount plagiarism attacks and completely break the system with minimal changes to documents.

A text sifting oracle does not reveal document fingerprints, but responds that a given document is an original or a plagiarism. An adversary with access to such an oracle is also a threat, because they can iteratively modify a document and find the minimum modification necessary to produce a false negative. This is not a problem if all users of the oracle are

trusted. Heintze [8] discusses methods for allowing public access to such an oracle.

An unaided adversary is one without access to document fingerprints, a text sifting oracle, or the secret key. An unaided adversary can only mount a successful plagiarism attack by making many changes in a document and hoping that the document fingerprint is significantly changed. In the cluster formation phase, an adversary can not predict the clusters which will be output and can only affect the clusters by changing many tokens. An adversary can make a large number of changes in a small region and have a high probability of changing the clusters from that region, but there is no way for the adversary to predict which clusters will be output by the sifting phase, so changes from the small region will probably not significantly change the document fingerprint. The only way to significantly change an entire document fingerprint is to change many tokens in all regions of the document.

A typical text sifting system will choose a similarity measure (usually S_3) and a threshold, and will decide that one of a pair of documents is a copy if the two have a similarity greater than the threshold. When an adversary attacks a document, one way to measure the distortion introduced by the attack is to consider the average similarity, taken over all keys, of the original document fingerprint and the attacked document fingerprint. We refer to this average similarity as the *attack similarity*. The intuition is that two documents with large attack similarity are likely to be very similar to human readers of the documents, for example for detecting plagiarism or copyright violations.

We consider an attack to be successful if the attack similarity is greater than the threshold but the similarity measured by the system is less than the threshold. An attack may result in an attack similarity that is below the threshold, but we do not consider this to be a successful attack because the attacker was forced to distort the document to the point where the attacked document no longer resembles the original document enough to constitute a copy.

The measured similarity is likely to be close to the attack similarity. This is made even more likely with the use of large document fingerprints. Importantly, an adversary can not know if the measured similarity will be higher or lower than the attack similarity. It is therefore difficult for an adversary to launch a successful attack; the attack must be strong enough that the attack similarity is above the threshold, yet the attacker must be lucky enough to have the measured similarity fall below the threshold.

From the perspective of an unaided adversary, the sifting stage of a text sifting system essentially performs a random selection on all clusters from the cluster formation stage. For both pure hashing and minhashing, if an attack causes c percent of clusters from the cluster formation phase to change, then we expect c percent of the cluster hashes in the document fingerprint to change as well. We will therefore consider the effects of various attacks by looking at how those attacks affect the clusters produced by the cluster formation stage.

3.1 Cosmetic Attacks

Cosmetic attacks are attacks that are eliminated in the preprocessing phase. When tokens are words, cosmetic attacks include changing capitalization, changing formatting, or adding spaces and extra punctuation. When tokens are letters, cosmetic attacks include changing capitalization, inserting punctuation, and substituting symbols for characters (for example, "| <" for "k" and "1" for "I"). Cosmetic attacks do not change the output of a text sifting function at all.

3.2 Scrambling Attacks

Scrambling attacks are attacks which add, delete, or change individual tokens. For example, when words are used as tokens, scrambling attacks correspond to adding words, deleting words, and changing words. Changing a word can mean anything from changing one letter in the word to substituting a new word; in either case, the token hash for the changed word will be completely different.

We will now consider, for several different cluster formation methods, the effects of scrambling attacks.

Sliding Window: Assume a sliding window of length w. Adding a token removes $w-1$ old clusters and adds w new clusters. Deleting a token removes w old clusters and adds $w-1$ new clusters. Changing a token removes w old clusters and adds w new clusters. Note that all these attacks all have approximately the same effect on the clusters that are output.

If an attacker performs a scrambling attack once every w tokens, they can virtually ensure that the sliding window never sees any of the same clusters it saw in the original document, and can therefore (barring hash collisions) ensure that every hash value in the output is different.

The exception to this occurs if an attack accidentally causes a cluster from the original document to appear in the attacked document. These accidents can occur for all three scrambling attacks. For example, when adding tokens, a token can be added next to an identical token, or can be added in such a way that one of the w new clusters is the same as a different cluster in the original document. Similar coincidences can occur when removing and changing tokens. The probability of such an occurrence is low, but after many attacks on many documents it does occur, as was shown in our experiments.

Random Skipping: Assume that all clusters produced by random skipping have a fixed length l, since fixed-length clusters were found experimentally to be better than variable-length clusters. Then, ignoring edge effects at the end of the document, a document with n words will have n clusters. Each cluster includes the head token and $l-1$ other tokens, for a total of $n(l-1)$ non-head tokens in clusters. Therefore, each token will be at the head of one cluster and will be in an average of $l-1$ other clusters.

Adding a token causes one cluster to be added, the cluster that starts with that token. No clusters will be removed, but clusters will change if they select this new token, so an average of $l-1$ clusters will change.

Deleting a token causes one cluster to be removed, the cluster that started with that token. Clusters will change if they formerly held the deleted token, so an average of $l-1$ clusters will change.

Changing a token is equivalent to deleting the token and adding a new one in its place. It does not cause any clusters to be added or removed, but it guarantees that the cluster that it heads will change. An average of $2(l-1)$ additional clusters will also be changed, for an average of $2l-1$ total clusters changed. Note that changing tokens has approximately twice the effect of adding or deleting tokens.

Random Partitioning: If a token is added, $\frac{1}{b}$ of the clusters will ignore it. If a token is removed, $\frac{1}{b}$ of the clusters were ignoring it anyway and will be unaffected. For both adding and removing tokens, the rest the clusters are affected according to the cluster formation mechanism being used, sliding window or random skipping, but the effect is multiplied by $b-1$. For example, if using random partitioning with

a sliding window, adding a token will result in the removal of $(w-1)(b-1)$ old clusters and the addition of $w(b-1)$ new clusters.

If a token is changed, there are two possibilities. If the replacement token is in the same partition as the original token, then as before the effect depends on the cluster formation mechanism being used but is multiplied by $b-1$. If the replacement token and the original token are in separate partitions, the effect is the same, but multiplied by b. Since we use $b = 2$, at least half of the time a changing attack will cause twice the normal effect. Random partitioning, like random skipping, is affected more when tokens are changed than when tokens are added or removed.

3.3 Large-Scale Attacks

Large-scale attacks are attacks where large numbers of contiguous tokens are manipulated. Attacks are considered to be large-scale when the edge-effects of the attacks are small compared to the main effects. We will discuss large-scale additions of tokens, large-scale deletion of tokens, and large-scale rearranging of tokens.

Large-Scale Additions: When a large amount of text is inserted into a location in the middle of a document, it will disrupt the clusters that used to span the location, and will generate new clusters that cross the boundary between the original tokens and the inserted tokens. We assume that these edge effects are small relative to the effects of the large number of new tokens in the document.

Now consider the non-edge effects of a large-scale addition. With pure hashing, all of the cluster hashes from the original document will be present, but there will be a large number of new cluster hashes in the the document fingerprint as well. Ignoring edge effects, the S_3 similarity from pure hashing will still be 1. When min-hashing with a constant output size, some of the cluster hashes from the original document will be forced out by cluster hashes from the added tokens. The fraction of original cluster hashes is expected to be equal to the size of the original document as a fraction of the size of the new, expanded document, for both pure hashing and min-hashing. For example, if the new tokens are double the size of the original document, then we expect $\frac{1}{3}$ of the cluster hashes in the document fingerprint to come from the original document.

Large-Scale Deletions: Similar to the case of large-scale additions, we assume that edge effects are small relative to the effects of the deletion of a large number of tokens.

With pure hashing, deleting a large number of contiguous tokens will result in the clusters which correspond to those tokens being removed. These non-edge effects mean that if a fraction f of a document is deleted, only $1-f$ of the original cluster hashes will remain in the document fingerprint. However, ignoring edge effects, the S_3 similarity will still be 1, because the attacked document will be contained within the original document. With a constant-sized min-hash, $1-f$ of the original cluster hashes will remain in the document fingerprint, and the rest will be replaced by new cluster hashes which come from the remaining portion of the text.

Large-Scale Rearranging: In the previous two cases we ignored edge effects. However, in the case of large-scale rearranging, edge effects are the only effects. If a large region of a document is moved to another spot in the same document, the only clusters that will change are those along the old and new boundaries of the region. Therefore, a small number of large-scale rearranging attacks will have relatively little effect on the document fingerprint with either pure hashing or min-hashing.

3.4 Experimental Results

As was mentioned before, Broder et al. [5] and Heintze [8] have demonstrated that these methods work well in the absence of adversaries. We now consider the robustness of text sifting functions against simple attacks.

Experimental Setup: Attacks were tested on a library of 100 articles from Reuters [3], approximately uniformly distributed in size from 1 kB to 6 kB. These articles were randomly chosen from a set of 9000 articles from Reuters. Both the attacks and the text sifting system were run in MATLAB [11]. MD5 was used as a hash function because a MATLAB implementation by Suter [18] was readily available, but it would generally be too slow to use in a production system. The results of the tests are shown in Table 2.

Words were used as tokens. We found that with random skipping it is best to have fixed-length clusters, so all clusters were formed of 10 tokens, both in sliding window tests and random skipping tests. When performing random skipping, clusters were accepted with a probability of 0.3, as this gave the best performance. We found that the optimal number of partitions for random partitioning was 2. The modulus for pure hashing was 10, so 1 out of every 10 clusters was output. For min-hashing, a constant 100 clusters were requested, although some documents sometimes fell slightly short of this.

Attacks: We tested eight attacks in two suites. The first suite was "intelligent" attacks and the second suite was random attacks. Both suites consisted of adding attacks, deleting attacks, changing attacks, and combination attacks.

The intelligent attacks consisted of attacking the tokens in positions $9, 19, 29, \ldots$. This ensured that every window of size 10 contained a single change. For the adding attack this meant adding tokens; for the deleting attack this meant deleting tokens; for the changing attack this meant changing tokens; and for the combination attack this meant alternating between adding, deleting, and changing tokens.

To make the tests equivalent, we made exactly as many random attacks of each type as there were intelligent attacks. For each document, if the document consisted of n tokens, the random adding attack added $0.1n$ random tokens to random locations in the document; the random deleting attack removed $0.1n$ tokens at random; the random changing attack changed a random set of $0.1n$ of the tokens into different, randomly-chosen tokens; and the random combination attack performed all three attacks on approximately $0.035n$ tokens each.

Results: Table 2 shows the results of the tests. The tests demonstrate one instantiation of a text sifting system, with a randomly-chosen key. We attacked all 100 documents in our library with the eight attacks and computed the S_1 and S_3 similarities between the original and attacked versions of each document for four different text sifting systems. The values in the table are the average similarities over all 100 documents.

The table shows that random skipping and random partitioning both offer significant advantages against the intelligent attacks. The intelligent plagiarism attacks succeed against a sliding window, but fail against systems using random skipping, random partitioning, or both. Against random attacks, the record is mixed. Random skipping and random partitioning do slightly better than the sliding window against adding and deleting attacks, but perform

worse on changing attacks, and by extension combination attacks, as explained in Section 3.2.

The data also illustrates the extra susceptibility of random skipping and random partitioning to changing attacks, as opposed to adding and deleting attacks. The combination attacks, which include all three scrambling attacks, lie somewhere in the middle.

Pure hashing generally performed better than min-hashing, which is surprising because the documents were small enough that only a quarter had fingerprints with size larger than 100, which was the size of fingerprints produced by min-hashing. What seems to happen is that on smaller documents min-hashing wastes space by storing too many clusters, while on longer documents min-hashing doesn't store enough clusters.

One interesting thing to note is that changing only 10% of the tokens in a document lowers the maximum similarity measurement to approximately 0.4. This happens because we used a large number of tokens per cluster. With long clusters, a single change in a document has the potential to affect many clusters. Using shorter clusters will help; using a length of 6 raised the maximum similarity measurement to approximately 0.65. However, decreasing the cluster length increases the number of false positives, so the length that is ultimately chosen depends on the needs of the specific system. A small window might work well with a database of papers, but will result in too many false positives in a full search of the internet.

Note also that the sliding window does not always result in similarities of zero when up against the intelligent attacks. This is the result of a small number of coincidences as explained in Section 3.2. They did not occur often, but they did occur occasionally, and that was enough to raise some of the averages above zero. However, those attacks only produced very small similarities (caused by one cluster matching) in a handful of documents over all the tests we performed.

False Positives: In our small library of 100 articles, for all three similarity measures in Section 2.4, all pairs of different documents had similarities of zero. Of course, all documents had a similarities of one with themselves. Occasionally in our tests (although not among our 100 articles), we found unrelated documents that shared a single cluster, but that occurrence was very rare. An S_3 similarity of even 0.1 generally arises only from plagiarism attacks, and an S_3 similarity of 0.2 almost certainly implies a plagiarism attack.

In a larger library, say all web pages on the internet, there is a significant probability of false positives. It is conceivable that two unrelated documents could end up with similar fingerprints, especially if the documents are short and have small fingerprints, and if they contain a lot of generic or boilerplate text. Having a large number of tokens per cluster and a large number of clusters per fingerprint will significantly decrease the probability of false positives. Both Broder et al. [5] and Heintze [8] address methods for reducing false positives.

A second source of false positives is attack-induced false positives. We do not consider these to be a problem because for a plagiarist, inducing a false positive is a loss.

4 Conclusions

Text sifting is an extension of previous work on document similarity detection, with a focus on plagiarism detection and resistance to attacks. We presented a system that can withstand a class of plagiarism attacks which are very successful against previous systems. Our system achieves this resilience by eschewing the use of sliding windows, instead using cluster formation methods which can not easily be exploited by attackers.

Our system also performs well against random attacks, although slightly worse than sliding window systems. A hybrid text sifting system could generate half of the fingerprint with a sliding window and half with random skipping and/or random partitioning. Compared to straight text sifting systems with the same fingerprint size, this would result in increased performance against random attacks, especially changing and combination attacks, but would result in reduced performance against intelligent attacks.

References

[1] Glatt plagiarism services. http://www.plagiarism.com.

[2] Plagiarism.org. http://www.plagiarism.org.

[3] Reuters articles. http://wim.fzi.de:8080/Reuters/Reuters.zip, from http://km.aifb.uni-karlsruhe.de/kaon2/frontpage.

[4] A. Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[5] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *6th International World Wide Web Conference*, 1997.

[6] A. Broder and M. Mitzenmacher. Completeness and robustness properties of min-wise independent permutations. In *RANDOM*, pages 1–10, 1999.

[7] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[8] N. Heintze. Scalable document fingerprinting. In *Second USENIX Electronic Commerce Workshop*, pages 191–200, 1996.

[9] P. Indyk. A small approximately min-wise independent family of hash functions. In *Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 454–456, 1999.

[10] C. Manning and H. Schtze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.

[11] MathWorks. Matlab. http://www.mathworks.com/.

[12] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.

[13] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algoriothms for document fingerprinting. In *SIGMOD 2004*, San Diego, CA, 2003.

[14] N. Shivakumar. *Detecting Digital Copyright Violations on the Internet*. PhD thesis, Stanford University, 1999. http://www-db.stanford.edu/s̄hiva/.

[15] N. Shivakumar and H. Garcia-Molina. Scam: A copy detection mechanism for digital documents. In *2nd International Conference in Theory and Practice of Digital Libraries*, Austin, Texas, 1995.

Pure Hashing								
	Slid. Win.		Rand. Skip.		Slid. Win. & Rand. Part.		Rand. Skip. & Rand. Part.	
	S_1	S_3	S_1	S_3	S_1	S_3	S_1	S_3
Intelligent adding attack	0	0	0.184	0.339	0.152	0.287	0.182	0.332
Intelligent deleting attack	0.00114	0.00264	0.187	0.354	0.157	0.296	0.190	0.346
Intelligent changing attack	0.000676	0.00148	0.0557	0.115	0.0482	0.0969	0.064	0.129
Intelligent combination attack	0	0	0.115	0.228	0.0914	0.180	0.109	0.211
Random adding attack	0.242	0.422	0.248	0.431	0.245	0.427	0.238	0.415
Random deleting attack	0.205	0.379	0.216	0.389	0.205	0.375	0.218	0.387
Random changing attack	0.194	0.347	0.109	0.213	0.127	0.243	0.106	0.203
Random combination attack	0.205	0.370	0.151	0.287	0.174	0.317	0.155	0.285

Min-Hashing								
	Slid. Win.		Rand. Skip.		Slid. Win. & Rand. Part.		Rand. Skip. & Rand. Part.	
	S_1	S_3	S_1	S_3	S_1	S_3	S_1	S_3
Intelligent adding attack	0	0	0.168	0.290	0.139	0.243	0.176	0.298
Intelligent deleting attack	0.000513	0.00102	0.176	0.302	0.148	0.258	0.171	0.292
Intelligent changing attack	0.000251	0.000500	0.0557	0.105	0.0403	0.0768	0.0550	0.104
Intelligent combination attack	0.000456	0.000901	0.106	0.191	0.0907	0.165	0.106	0.190
Random adding attack	0.222	0.362	0.226	0.370	0.232	0.375	0.223	0.364
Random deleting attack	0.200	0.3322	0.213	0.355	0.205	0.342	0.206	0.339
Random changing attack	0.213	0.349	0.104	0.188	0.132	0.230	0.110	0.196
Random combination attack	0.178	0.300	0.147	0.254	0.155	0.266	0.141	0.245

Table 2: The effects of scrambling attacks on text sifting.

[16] N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *1st ACM Conference on Digital Libraries*, Bethesda, Maryland, 1996.

[17] D. Stinson. *Cryptography: Theory and Practice*. CRC Press, 1995.

[18] Hans-Peter Suter. Matlab md5 implementation. http://www.mathworks.com/matlabcentral/fileexchange.