The attached DRAFT document (provided here for historical purposes) has been superseded by the following publication:

Publication Number: **NIST Special Publication 800-168**

Title: **Approximate Matching: Definition and Terminology**

Publication Date: **05/31/2014**

- Final Publication: http://dx.doi.org/10.6028/NIST.SP.800-168
  (which redirects to:
  http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-168.pdf).

- Information on other NIST Cybersecurity publications and programs can be found at: http://csrc.nist.gov/

DRAFT NIST Special Publication 800-168

Approximate Matching: Definition and Terminology

Frank Breitinger
Barbara Guttman
Michael McCarrin
Vassil Roussev

http://dx.doi.org/10.6028/NIST.SP.xxx

National Institute of Standards and Technology
U.S. Department of Commerce

DRAFT NIST Special Publication 800-168

Approximate Matching: Definition and Terminology

Frank Breitinger
Hochschule Darmstadt University of Applied Sciences

Barbara Guttman
Software and Systems Division
Information Technology Laboratory

Michael McCarrin
Naval Postgraduate School

Vassil Roussev
University of New Orleans

http://dx.doi.org/10.6028/NIST.SP.xxx

January 2014

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

Authority

This publication has been developed by NIST to further its statutory responsibilities under the Federal Information Security Management Act (FISMA), Public Law (P.L.) 107-347. NIST is responsible for developing information security standards and guidelines, including minimum requirements for Federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate Federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), *Securing Agency Information Systems*, as analyzed in Circular A-130, Appendix IV: *Analysis of Key Sections*. Supplemental information is provided in Circular A-130, Appendix III, *Security of Federal Automated Information Resources*.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on Federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other Federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-168
Natl. Inst. Stand. Technol. Spec. Publ. 800-168, 14 pages (January 2014)
http://dx.doi.org/10.6028/NIST.SP.xxx
CODEN: NSPUE2

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by Federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, Federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. All NIST Computer Security Division publications, other than the ones noted above, are available at http://csrc.nist.gov/publications.

Public comment period: *January 27, 2014 through March 21, 2014*

National Institute of Standards and Technology
Attn: Software and Systems, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8970) Gaithersburg, MD 20899-8970
Email: match@nist.gov

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in Federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Acknowledgements

This document is based on the work of the NIST Approximate Matching Working Group, which consists of Frank Breitinger, John Delaroderie, Simson Garfinkel, Barbara Guttman, John Kelsey, Jesse Kornblum, Mary Laamanen, Michael McCarrin, Vassil Roussev, Clay Shields, John Tebbutt, Douglas White, and Joel Young.

Conformance Testing

This publication has been developed by NIST to further its statutory responsibilities under the Federal Information Security Management Act (FISMA), Public Law (P.L.) 107-347. NIST is responsible for developing information security standards and guidelines, including minimum requirements for Federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate Federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), Securing Agency Information Systems, as analyzed in Circular A-130, Appendix IV: Analysis of Key Sections. Supplemental information is provided in Circular A-130, Appendix III, Security of Federal Automated Information Resources.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on Federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other Federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

Table of Contents

Introduction

1.1 Authority

This publication has been developed by NIST to further its statutory responsibilities under the Federal Information Security Management Act (FISMA), Public Law (P.L.) 107-347. NIST is responsible for developing information security standards and guidelines, including minimum requirements for Federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate Federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), Securing Agency Information Systems, as analyzed in Circular A-130, Appendix IV: Analysis of Key Sections. Supplemental information is provided in Circular A-130, Appendix III, Security of Federal Automated Information Resources.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on Federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other Federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

1.2 Purpose and Scope

Approximate matching is a promising technology for designed to identify similarities between two digital artifacts. It is used to find objects that resemble each other or to find objects that are contained in another object. This can be very useful for filtering data for security monitoring, digital forensics, or other applications.

1.3 Audience

The intended audience of this document is security digital forensics programmers and other technical professionals with a need to determine, build, or use technology to identify similarity.

2. Definition and terminology

Approximate matching is a generic term describing any technique designed to identify similarities between two digital artifacts. In this context, an *artifact* (or an *object*) is defined as an arbitrary byte sequence, such as a file, which has some meaningful interpretation.

Different approximate matching methods may operate at different levels of abstraction. At the lowest level, generic techniques may detect the presence of common byte sequences (substrings) without any attempt to interpret the artifacts. At higher levels, approximate matching can incorporate more abstract analysis that is closer to what a human analyst might do. The overall expectation is that lower level methods would be faster, and more generic in their applicability, whereas higher level ones would be more targeted and require more processing.

One common approach in security and forensic analysis is to find identical objects using cryptographic hashing. Approximate matching can be viewed as a generalization of that idea in that, instead of providing a yes/no {0, 1} answer to a comparison, it provides a range of outcomes, [0, 1], with the result interpreted as a measure of similarity.

2.1 Use cases

Broadly, there are two types of similarity queries that are of interest – *resemblance* and *containment* [1]. In the case of resemblance, we compare two similarly sized objects and interpret the result as a measure of the commonality between them; for example, two successive versions of a piece of code are likely to resemble each other substantially. When the compared objects differ in size significantly, such as a file and a whole-disk image, the test for commonality is interpreted as a *containment* query because it addresses the question of whether the large object contains the smaller one.

An approximate matching algorithm should be able to handle at least one of the following challenges (divided according to whether the query type is (R)esemblance or (C)ontainment) [2, 3]:

Object similarity detection (R): identify related artifacts, e.g., different versions of a document.

Cross correlation (R): identify artifacts that share a common object, e.g., a Microsoft Word document and a PDF document containing the same image, or other embedded object.

Embedded object detection (C): identify a given object inside an artifact, e.g., an image within a compound document or an executable inside a memory capture.

Fragment detection (C): identify the presence of traces/fragments of a known artifact, e.g., identify the presence of a file in a network stream based on individual packets.

In most analytical scenarios, approximate matching is used to *filter* data in, or out, based on a known reference set. In security monitoring applications, approximate matching could potentially be used to *blacklist* known bad artifacts, and (by extension) anything closely resembling them. However, approximate matching is not nearly as useful when it comes to *whitelisting* artifacts as malicious content can often be quite similar to benign content; e.g., a backdoored **ssh** server would look very similar to a regular one.

2.2 Terminology

Although the common language definition of 'similarity' is sufficient to give an intuitive sense of the term, the multitude of ways in which two artifacts can be said to be similar poses a challenge when attempting to describe the purpose and behavior of approximate matching algorithms. For example, two strings 'ababa' and 'cdcdc' might be considered similar in that they both have five characters ranging over two alternating values, or they might be treated as dissimilar because they have no common characters. To resolve this ambiguity, approximate matching algorithms define similarity in terms of *features* that represent the characteristics of the artifacts pertinent to the algorithm's method of comparison.

> *Features.* Features are the basic elements through which artifacts are compared. Comparison of two *features* always yields a binary {0, 1} outcome indicating a match or non-match; because features are defined as the most basic comparison unit that the algorithm considers, partial matches are not permitted. Generally, a *feature* can be any value derived from an artifact. Each approximate matching algorithm must define the structure of its features and the method by which they are derived. For example, an algorithm might define a feature as a (byte, offset) pair produced by reading the value of a byte and storing it along with the offset at which it was read.
>
> *Feature set.* The set of all features associated with a single artifact is its feature set. Each algorithm must include a criteria by which candidate features are selected for inclusion in this set. For example, an algorithm might select all the (byte, offset) pairs produced by reading every 16th byte in the artifact.
>
> *Similarity.* The similarity of two artifacts, as measured by a particular approximate matching algorithm, is defined as an increasing monotonic function of the number of matching features contained in their respective feature sets.

Based on the level of abstraction of the similarity analysis performed, approximate matching methods can be placed in one of three main categories [4]:

> *Bytewise* matching relies only on the sequences of bytes that make up a digital object, without reference to any structures within the data stream, or to any meaning the byte stream may have when appropriately interpreted. Such methods have the widest applicability as they can be applied to any piece of data; however, they also carry the implicit assumption that artifacts that humans perceive as similar have similar byte-level encodings. The validity of this assumption varies widely and the analysts must have the appropriate background to interpret the results correctly.
>
> *Syntactic* matching uses internal structures present in digital objects. For example, the structure of a TCP network packet is defined by an international standard and matching tools can make use of this structure during network packet analysis to match the source, destination or content of the packet. Syntax-sensitive similarity measurements are specific to a particular class of objects that share an encoding but require no interpretation of the content to produce meaningful results.

Semantic matching uses contextual attributes of the digital object to interpret the artifact in a manner that more closely resembles human cognitive processing. For example, perceptual hashes allow the matching of visually similar images and are unconcerned with the low-level details of how the images are persistently stored. Semantic methods tend to provide the most specific results but also tend to be the most computationally expensive ones.

In current literature, researchers use a number of terms to refer to various approximate matching methods: *fuzzy hashing* and *similarity hashing* denote bytewise approximate matching; *perceptual hashing* and *robust hashing* denote semantic approximate matching. There is no widely-used pre-existing terminology for syntactic approximate matching as it is mostly viewed as pre-processing (to separate the features) before hashing, or applying a bytewise approximate matching algorithms. For example, network flows are usually reconstructed before any processing is done on them.

Bytewise approximate matching algorithms work in two phases. In the first, a *similarity digest* representation (also referred to as a *signature* or *fingerprint*) is generated from the original data. In the second phase, digests are compared to produce a *similarity* score. More precisely:

Similarity digest. A similarity digest is a (compressed) representation of the original data object's feature set that is suitable for comparison with other similarity digests created by the same algorithm. In most cases, the digest is much smaller than the original artifact and the original object is not recoverable from the digest.

Every bytewise approximate matching technique requires at least two core functions:

Feature extraction function: identifies and extracts features/attributes from each objectk, allowing a compressed representation of the original object. The mechanism by which features are picked and interpreted depends on the approximate matching algorithm. The representation of this collection is the *similarity digest* of the object.

Similarity function: compares two similarity digests and outputs a score. The recommended approach is to assign a score s in the $0 \leq s \leq 1$ range, where 0 indicates no similarity and 1 indicates high similarity. This score represents a normalized estimate of the number of matching features in the feature sets corresponding to the artifacts from which the similarity digests were created.

Normalization strategy: The similarity function can follow one of two normalization strategies, depending on whether the algorithm describes resemblance or containment. For resemblance queries, the number of matching features will be weighed against the total number of features in both objects. In the case of containment queries, the algorithm may disregard unmatched features in the larger of the objects' two-feature set.

Because features and feature sets can be arbitrarily complex and, furthermore, deal with byte-level structures to which meaning is not clearly assigned, the interpretation of the similarity score can prove challenging. To address this problem, some approximate matching algorithms make use of an empirically determined *threshold* value to attempt to correlate bytewise similarity scores with higher-level properties of interest. In such cases, the similarity score can be treated as a *confidence score*, where results above the threshold value are considered likely to exhibit common human-recognizable traits.

2.3 Essential requirements

Like traditional hash functions, there are several defining characteristics that approximate matching functions should exhibit. Each algorithm should define how it incorporates each of these properties and how it satisfies the reporting requirements for those properties, where appropriate.

Similarity preservation: Similarity digests must be constructed such that the outcome of a comparison between any two digests is uniquely determined by the similarity of the artifacts from which they were produced. That is, if A' is a similarity digest created from artifact A and B' is a similarity digest created from artifact B, the results of comparing A' and B' should be uniquely determined by the similarity of A and B.

Self-evaluation: The similarity measure should be accompanied by a measure of the accuracy of the matching technique under the circumstances in which it is used, e.g., a margin of error or confidence level. The description of the output score should also state whether a score of 1 indicates an exact match.

Compression: A compact similarity digest is desired as it normally allows a faster comparison and requires less storage space. In the best case, it will have a fixed length like the output of traditional hash functions. If the efficiency and reliability of the results remains unchanged, then a shorter similarity digest is preferable.

Ease of computation: First, the algorithm description should include the results of testing the runtime efficiency of the feature extraction function and of the similarity function. The former might be expressed relative to a standard hashing algorithm, such as SHA-1.

Second, the algorithm description should state the theoretical complexity for a similarity digest comparison which is known as O-notation. For instance, common lookup complexities for comparing a single digest against a database with n entries, are:

$O(1)$	search of cryptographic hash values stored in hash tables (e.g. dictionaries)
$O(log_2\ n)$	cryptographic hash values stored in binary trees or a sorted list
$O(n)$	cryptographic hash values stored in an unsorted list, or another kind of search in which no indexing or sorting is possible

2.4 Reliability of results

The reliability of the results for a given approximate matching technique depends on three factors. Each algorithm should define how it incorporates these factors and how it satisfies their reporting requirements.

Sensitivity & robustness: The algorithms should provide some measure of their robustness. A technique's robustness will define the operating conditions in which it can function effectively, also called its *performance envelope*. For example, robustness addresses the minimum and maximum object sizes that an algorithm can reliably distinguish between.

Precision & recall: The algorithms should include a description of the methods used to determine its reliability and to select the test data. Specifically, it should indicate whether test data is culled from existing collections or developed solely to specifically support testing. Test results may include precision & recall rates as well as false positive and false negative rates.

Security of results: The algorithms should indicate whether they include security properties designed to prevent attacks. Such attacks include manipulation of the matching technique or input data such that a data object appears dissimilar to another object to which it is similar or similar to another object with which it has little in common.

3. Standardized testing for bytewise approximate matching

Currently, algorithm developers use different methods and test data to evaluate approximate matching algorithm performance[3]. The remainder of this discussion focuses on putting forth a set of tests that can be used to characterize approximate matching methods. These are not a definitive set, but demonstrate various attributes that can be tested and some approaches for doing so.

3.1 Efficiency

There are at least three basic types of efficiency for which algorithms should be evaluated:

Generation efficiency. Generation efficiency measures the throughput rate of an algorithm while it processes the raw input to produce the similarity digest. To enable useful comparisons across different architectures, it is recommended that the throughput rate of a standard algorithm implementation, such as SHA-1 in *openssh*, be included as a reference point.

Comparison efficiency. The comparison efficiency measures the rate at which similarity digest comparisons can be executed. It is useful to have both a formal analysis, which provides the theoretical complexity of the comparison (in O-notation) and an empirical evaluation based on a reference data set.

Another evaluation aspect is the ability of the technique to efficiently utilize parallel computational resources; these may include conventional multi-core CPU architectures, as well as massively parallel ones, such as GPUs. To that end, tests should include scalability analysis, which shows speedup as a function of available hardware concurrency.

Space efficiency. Traditional hash functions return a fixed length fingerprint; in contrast, the length of similarity digests is sometimes variable and proportional to the input length. If the digest is of variable length, space efficiency measures the ratio between input and the digest and returns a percentage value. More precisely,

$$\text{space efficiency} = \frac{\text{digest length}}{\text{input length}} \quad (1)$$

3.2 Sensitivity and robustness

Sensitivity is a measure of the ability of an approximate matching algorithm to find correlations among objects based on fine-grain commonality–the smaller the features being correlated, the more sensitive the algorithm is. Clearly, there is a threshold below which the sensitivity will be too high and all objects will appear similar; it is up to the algorithm designer to identify that threshold and incorporate it into the implementation.

Robustness is a measure of the ability of an approximate matching algorithm to find correlation among related objects in the presence of noise and routine transformations. Common transformations include fragmentation (e.g., during network transmission) and misalignment (adding content during artifact editing).

The following four tests (later called *modifications*) evaluate sensitivity & robustness for bytewise approximate matching algorithms: *fragment detection*, *single-common-block correlation*, *alignment robustness*, and *white noise resistance*. The first two are aimed at evaluating sensitivity, whereas the latter two measure robustness.

For the purposes of this discussion, we refer to each modification by the combination of a test name and parameter, e.g., 'fragment at 5%' or 'alignment at 4 KiB'. We denote as the examples indicate, the test parameter may be expressed as either an absolute or a relative value. In most cases, relative values tend to produce results that are more useful, but absolute values are particularly useful in alignment tests. In the follow the term option for this combination of a modification and a specific setting/test.

> *Fragment detection.* Fragment detection quantifies the length of the shortest sample from a data object, for which the similarity tool reliably correlates the sample and the whole object. Common uses include correlating a disk block, or network packet to file.

Therefore, it sequentially cuts $X \in$ {25%, 50%, 60%, 70%, 75%, 80%, 85%, 90%, 95%, 96%, 97%, 98%, 99%} of the original input and compares both inputs.

To simulate real-world scenarios by which fragments are created, two different cutting modes are suggested:

1. *Random cutting:* The test randomly decides at each step whether to cut at the beginning or the end of an input.
2. *End side cutting:* The test only cuts blocks at the end of an input.

   (Cutting from the beginning yields similar to the alignment test.)

> *Single-common-block correlation.* The single-common-block correlation test is designed to characterize the behavior of an algorithm in the case where two files share a single common object. That is, given two files f_1 and f_2 that share a common object O (but are otherwise dissimilar), what is the smallest O for which the similarity tool reliably correlates the two targets?

The test can be performed in controlled conditions as follows (parameters can be varied as necessary). First, two (pseudo-)random files f_1 and f_2 of size $S \in$ {512, 2048, 8192} KiB are created followed by the common block $O \in$ {75%, 50%, 40%, 30%, 20%, 10%, 5%, 4%, 3%, 2%, 1%} of S. Next, O overwrites f_1 and f_2 at different and randomly chosen offsets (the size of f_1 and f_2 remains equal to S and constant over time). Finally, we perform a comparison of f_1 and f_2. If we obtain a match score greater than zero, we reduce O further and repeat the process. To obtain statistically significant results, the location and content of the fragment is varied over multiple runs.

> *Alignment robustness.* Alignment robustness is an attempt to quantify the sensitivity of an algorithm to different alignments of the common data. Specifically, the test analyzes the impact of inserting byte sequences of size X at the beginning of an input, where the size of the sequence may be expressed in absolute, or relative terms.

1. *Fixed blocks:* Suggested parameter values for X : {1, 2, 3, 4, 8, 16, 32, 64} KiB. These cover the most common cases; also, the observed behavior tends to be periodic relative to the size of the modification. In

other words, testing intermediate parameters like {5, 6, 7} KiB do not produce unique scenarios.

2. *Relative blocks:* Suggested parameter values for X : {10%, 25%, 50%, 75%, 100%, 200%, 400%}; these numbers simulate changes on a larger scale.

White noise resistance. This test measures the amount of (uniformly) random noise that can be added to an object before the approximate matching algorithm becomes unable to correlate the original and the modified version. For example, for **ssdeep** [5] it was shown that a few changes distributed over the input are sufficient to prevent a match [6].

A random change is simulated by applying typical edit operations (namely insertion, deletion, and substitution) where each edit operation is chosen with the same probability. Additionally, each byte in the input is equally likely to be changed.

First, the original f_1 is copied to have f_2. Next, the test obfuscates f_2, i.e., X % of f_2 's bytes are manipulated where $X \in$ {0.1%, 0.25%, 0.5%, 0.75%, 1.0%, 1.5%, 2.0%, 2.5%}. (The range could be expanded but in actual testing no existing algorithm is able to correlate the original and the modified version if 2.5%, or more, of the bytes were manipulated.

3.3 Testing approximate-matching

Conceptually, there are two types of data that can be used to evaluate approximate matching algorithms-controlled (synthetic) data [7] and real data. The main advantage of controlled data experiments is that ground truth is constructed and, therefore, precisely known. This allows randomized tests to be run completely automatically and the results to be interpreted with standard statistical measures.

The obvious downside is that much of real data is far from random so the applicability of the result to the general case remains uncertain. Nevertheless, running controlled tests is quite useful in characterizing the baseline capabilities of different algorithms. Indeed, the results provide the necessary context for interpreting algorithm behavior on real data.

The obvious advantage of using real data is that the results can be directly be related to observable artifacts. However, the challenges of defining a representative sample, establishing the ground truth, and running experiments at scale (without a human in the loop) are non-trivial.

After surveying prior work in the field, we suggest that results from the two approaches are complementary and both should be considered in the evaluation process. The next two sections address the use of controlled and real world data.

Testing with controlled data. The main purpose of controlled data experiments is to know exactly the ground truth by carefully constructing the test cases. In this case, the goal is to build artifacts – files – that have known levels of commonality in the form of common substrings. The most practical way to accomplish this is to use (pseudo-)random data.

The first step is to determine the appropriate sizes for the constructed files. Based on a survey of the distribution of almost 1 000 000 file sizes in the **govdoc**-corpus[1], it is suggested that evaluating algorithms at six reference file sizes–1, 4, 16, 64, 256 and 1024 KiB–would provide a representative sample. As shown in Table 1, nearly 91% of all files are smaller than 1 MiB.

Table 1. Cumulative empirical file size distribution in the **govdoc**-corpus.

File size range (KiB)	≤ 4	≤ 16	≤ 64	≤ 256	≤ 1024
Cumulative probability (%)	5.4	20.71	52.54	75.82	90.60

Test methodology. The proposed approach is conceptually simple and consists of four basic steps: build a set of unique files, create mutations of them using one of the four modification methods presented in Sec. 2.2, run the approximate matching comparisons between original and modified version (for all algorithms), and summarize the results with appropriate statistics.

> For every choice of file size and modification method, each test has two additional parameters: *file count* and *number of runs*. The former specifies the number of files in the test set; the latter specifies the number of independent test runs to be executed (where each run creates its own new test set).
>
> In terms of execution time, having a set of *file-count* files results in $\textit{file-count}^2$ comparisons. Hence, the total number of comparisons per algorithm is calculated by $\textit{file-count}^2 \times \textit{runs} \times o$ where o is the number of all options.

Test set manipulations: The mutated set is created by applying the four generic modification techniques from Sec. 2.2. Specifically, the following parameter set- tings are recommended:

Fragment detection: f_2 is a fragment of f_1 where the size of f_2 is X % of the size of f_1 , where X = {1%, 2%, 3%, 4%, 5%, 10%, 15%, 20%, 30%, 50%}. (The fragment is chosen randomly across runs.)

Single-common-block correlation: f_1 and f_2 have equal size and share a common byte string (block) of size X = {1%, 2%, 3%, 4%, 5%, 10%, 15%, 20%, 30%, 50%}.

> (The position of the common block, and its content are chosen randomly for each file/run combination.)

Alignment robustness: f_2 is a copy of f_1, prefixed with a random byte string of length X = {1%, 2%, 3%, 4%, 5%, 10%, 20%}. (Content of the prefix is randomized across runs.)

Random-noise resistance: f_2 is an obfuscated version of f_1, i.e., X % of f_2 's bytes are edited, where X = {0.5%, 1.0%, 1.5%, 2.0%, 2.5%} of the file size.

To sum up, there are 29 different options for the controlled data test.

Testing with real data. As already mentioned, two of the main challenges in testing with real data are the choice of representative samples, and the establishment of ground truth. The

---

[1]"These documents were obtained by performing searches for words randomly chosen from the Unix dictionary, numbers randomly chosen between 1 and 1 million, and randomized combinations of the two, for documents of specified file types that resided on web servers in the .gov domain using the Yahoo and Google search engines" (http://digitalcorpora.org/corpora/files).

former is outside the scope of this discussion, as the choice would depend, to some degree, on the expected characteristics of the target data. For example, for general evaluation of artifacts found on the Internet, the **govdocs**-corpus is a good starting point.

The focus of this section is to provide an approach for establishing ground truth using automated means. The proposed approach is to use the longest common substring (LCS) as the reference metric and to characterize the behavior of bytewise approximate matching algorithms with respect to this metric.

Using a string comparison algorithm as a reference is a natural choice given that the algorithms treat the data objects as plain strings with no attempt to parse or interpret them. LCS should be considered a first-order approximation as two objects may have a lot more in common than what the LCS result suggests, so further refinements are to be expected at a later stage.

Given an unordered pair of files (f_1, f_2), define the absolute (L_a) and relative (L_r) results as follows:

$$L_a = LCS(f_1, f_2), \text{ where } 0 \leq L_a \leq min(|f_1|, |f_2|). \quad (2)$$

$$L_r = \lceil L_a / min(|f_1|, |f_2|) \rceil, \text{ where } 0 \leq L_r \leq 1. \quad (3)$$

where $|f|$ denotes the file size in bytes.

Broadly, any two strings sharing a substring are related; however, we suggest a more practical lower bound on the minimum amount of commonality to declare two files related. Specifically, we require that the absolute size L_a is at least 100 (bytes) and that the relative result L_r exceeds 0.5% of the size of the smaller file. More formally, the true positive function $TP_{lcs}(f_1, f_2)$ is defined as

$$TP_{lcs}(f_1, f_2) \equiv L_a \geq 100 \wedge L_r \geq 1 \quad (4)$$

(Note: result of L_r is rounded and thus 0.5 is equal to 1.)

Clearly, the true negative function $TN_{lcs}(f_1, f_2) = \neg TP_{lcs}(f_1, f_2)$.

Approximate ground truth LCS is a well-studied problem and has known solutions of quadratic time complexity–$O(mn)$, where m and n are the string lengths. Given that files could be quite large, the exact solution quickly becomes too burdensome to be practical. Therefore, we suggest an approximation of the longest common substring which, by design, provides a lower bound on LCS; details are given in Appendix A.

References

1. A. Z. Broder. "On the resemblance and containment of documents," in *In Compression and Complexity of Sequences (SEQUENCES'97)*. IEEE Computer Society, 1997, pp. 21–29.
2. V. Roussev. "An evaluation of forensic similarity hashes," *Digital Forensic Research Workshop*, vol. 8, pp. 34–41, 2011.
3. F. Breitinger, G. Stivaktakis, and H. Baier. "FRASH: A framework to test algorithms of similarity hashing," in *13th Digital Forensics Research Conference (DFRWS'13)*, Monterey, August 2013.
4. F. Breitinger, H. Liu, C. Winter, H. Baier, A. Rybalchenko, and M. Steinebach. "Towards a process model for hash functions in digital forensics," *5th International Conference on Digital Forensics & Cyber Crime*, September 2013.
5. J. Kornblum. "Identifying almost identical files using context triggered piecewise hashing," *Digital Forensic Research Workshop (DFRWS)*, vol. 3S, pp. 91–97, 2006.
6. H. Baier and F. Breitinger. "Security aspects of piecewise hashing in computer forensics," *IT Security Incident Management & IT Forensics (IMF)*, pp. 21–36, May 2011.
7. F. Breitinger, G. Stivaktakis, and V. Roussev. "Evaluating Detection Error Trade-offs for Bitwise Approximate Matching Algorithms," *5th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, September 2013.
8. L. C. Noll. (2001) Fowler / Noll / Vo (FNV) Hash. [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv/index.html

Appendix

Approximate longest common substring

The basic idea of the approximate longest common substring metric (aLCS) is not to compare files byte by byte but rather block by block. To identify the blocks, we apply the rolling hash from **ssdeep**. Our settings aim at having blocks of ≈ 80 bytes. Instead of comparing blocks bytewise, each one is hashed and compared using the 64-bit FNV-1a hash [8]. Besides the hash value, we also store the entropy and length for each block in a final linear list called *alcs-digest*; a reference implementation is publicly available.[2]

Let L_a denote the absolute longest common substring of two alcs-digests. Comparing two alcs-digests is equal to comparing two linear lists. If the hash of an item on list *A* has the same value as the hash of an item on list *B*, we are convinced that L_a is greater than or equal to the length of the blocks corresponding to the hashes. If two consecutive items on list *A* have the same hash values as two consecutive items on list *B*, we sum up the length of both blocks to receive L_a. Of course, the usage of hash functions implies the possibility of false positives. Nevertheless, this is an easy and fast method to get a good estimation of the longest common substring.

Implementation details. The tool is implemented in C and separated into three steps: reading, hashing and comparing, which are declared in the main function. As it is a command line tool, it can be executed by **./aLCS <dir>** .

First, all files in **dir** are read. Out of the file names, we create "hash-tasks" which are added to a thread pool. A hash-task contains the path to a file and denotes "hash file". Depending on the number of threads, these tasks are processed. Once all alcs-digests are created, we perform an all-against-all comparison. Therefore, we create compare-tasks (compare $file_1$ against $file_2$) which are again added to the thread pool. The output is printed to the standard output.

The reference implementation has three main settings configurable in **header/config.h. MIN_LCS** is the minimum L_a length which is printed to **stdio** and is by default 0 (all comparisons are printed). The **THREAD_POOL_QUEUE_SIZE** is the length of the queue and should be $^{\text{fileamount} \times (\text{fileamount} - 1)}/_{2}$. **NUMTHREADS** is the number of threads which should be equal to the number of cores.

Verification of ground truth. To verify the correctness of our approximate longest common substring, we compared the results against LCS for a subset of *t5*. In order to do this, we implemented a parallelized LCS tool written in C++.[3] The output is a summary file structured similarly to our aLCS output: **file1 | file2 | LCS**. A small, ruby script is used to compare the LCS- summary and aLCS-summary.

Our subset consists of 201 randomly selected files. We compare these files using aLCS as well as LCS and finally compare both summaries. All $\frac{(200)\text{x}(201)}{2} = 20,100$ comparisons yield alcs scores in the correct range, *i.e.*, $0 \leq alcs \leq lcs$.

[2] https://www.dasec.h-da.de/staff/breitinger-frank/#downloads (last accessed 2013-05-09).

[3] https://www.dasec.h-da.de/staff/breitinger-frank/#downloads (last accessed 2013-05-09).

We also consider the distribution of the differences between the LCS and aLCS scores. Specifically, we define d_r for files f_1 and f_2 as follows:

$$d_r = \left\lceil \frac{lcs(f_1,f_2) - alcs\,(f_1,f_2)}{\min(|f_1|,|f_2|)} \right\rceil, d_r \in 0, 1\ldots,100.$$

In other words, we consider the score difference relative to the size of the smaller of the two files, and build the empirical distribution in Table 2. As we can see, upwards of 95% of the observed differences do not exceed 3% of the size of the smaller files – we consider this a reasonable starting point for our purposes (further research may refine this). If anything, this should give tools a slight boost as the available commonality would be underestimated.

Table 2. Empirical probability distribution function (*pdf*) and cumulative distribution function (*cdf*) for d_r.

X	0	1	2	3	4	5	10	15	20
$P_r\{d_r = X\}$	0.8869	0.0449	0.0155	0.0040	0.0047	0.0116	0.0062	0.0001	0.0000
$P_r\{d_r \leq X\}$	0.8869	0.9318	0.9473	0.9513	0.9561	0.9677	0.9834	0.9992	0.9999