Method for Approximate Content Discovery

Philippe Ombredanne, AboutCode.org

Abstract

This publication describes a new and improved combination of techniques and algorithms that form a method for efficient approximate digital content discovery and identification at scale.

This proposed method for matching approximate digital content and text fragments consists of these three elements:

1. Content-defined chunking algorithm to split content into fragments and select a subset of these fragments

2. Fingerprinting fragments with a locality sensitive hashing (LSH) function for approximate matching of these fragments. This fingerprint can embed multiple precision in a single bit string to tune the search precision and organize the search in rounds of progressively increasing precision.

3. Indexing-time approximate matching for deduplication where each new content fragment is added to the index if it is not already matchable approximately in the index, avoiding large duplications of content entries.

This publication describes the general context and problems this method attempts to resolve, and describes the method itself. It is completed by examples of applications with AI-Generated Code Search and Software Origin Discovery for Software Supply Chain Security. The methods described here have other applications beyond these including training data content deduplication for AI and LLMs.

Table of Contents

Problem

Content similarity matching for origin identification consists in these steps:
1. Building an index of existing, reference content using fingerprints of content and content fragments to accommodate large volume of content.
2. Matching content under analysis by computing similar fingerprints of content and content fragments. These fingerprints are then used to query the index.

At a high level, this is a "classical" Information Retrieval (IR) problem, that could resolved using classical search engine and IR techniques. Yet these techniques are ill suited because of the scale of the problem that is quite different from the scale that even the largest search engine handle. In a search engine such as Google, the index is extremely large, eventually addressing a significant part of the accessible web. But the queries of a search user are limited to 32 words. In contrast, the index for a content similarity system may be smaller, but the query -- the content under analysis to determine its origin using similarity as a proxy for origin -- is vastly larger, and may represent gigabytes of text. To understand the problem better, querying for gigabytes of text in a traditional search engine would imply submitting million or billions of queries for a small content similarity analysis, and the execution time, and the amount of resources required to accomplish this task would be prohibitively expensive.

Because of this, existing IR text indexing solutions do not work for content origin detection at scale.
The research has therefore evolved to pursue different approaches commonly referred as "Local Text Reuse Detection", "Plagiarism Detection".

The current state of the art consists either in:

1. A content-defined chunking algorithm such as as [AIKEN] winnowing used to break a content file in fragments
2. Computing a checksum on each fragment, using function such as a CRC32, MD5 or SHA checksum.

or:

1. Computing an approximate, fuzzy fingerprint on the whole content, using function such as [BRODER] Minhash or similar.

These existing matching approaches only match whole files approximately or fragments exactly; matching content fragments exactly has a high precision with low recall results, as it is not possible to find neighboring matches with an exact match. Furthermore, exact matching requires indexing the largest possible number of fragments to avoid false negative matches, and this increasing volume leads to more false positives. This exact matching does not work for several use case such as AI-generated content where each generation may have small variations given the same input prompt. And can be attacked by an adversary by making small content variations that will make the exact matches fail.

Solution

Our method for efficient content matching consists of three elements:
1. Chunking using a content-defined chunking algorithm to split content in fragments. This is used to break a content file in fragments considering the content, such that the same fragment boundaries are always selected when the same fragment is present. We consider alternative to the common winnowing.
2. Fingerprinting with a locality sensitive hashing (LSH) function for approximate matching of fragments. This fingerprint embeds multiple precision in a single bit string to tune the search precision and organize the search in rounds of progressive increasing precision.
3. Matching at indexing time where each new fragment added to the index if and only if it is not matchable in the index, avoiding large duplications of index entries.

Challenges and Method summary

Approximate content matching key challenges are 1) scale and 2) content duplication.

1) Scale is an issue not only because of the volume of content to search (potentially multi petabytes) but also because the query against this content is a gigabytes-size content set that is under analysis for third-party origin. In contrast, a search engine such as Google search limits queries to only 32 words, e.g, just a few bytes. Therefore practical content matching demands different approach es. In particular it is not practical to reuse a traditional IR (Information retrieval) inverted index because of the size of the analyzed content ; instead this demands a significant reduction of dimensionality of the problem.

State-of-the-art uses checksums as a proxy for content to achieve a reduction of the size of data processed. It breaks file content in content fragments (aka. snippets or chunks) using a content-defined heuristic, and computes a checksum on each snippet. The same checksums are collected from the analyzed content and are matched exactly against the index. This approach demands an ever growing index to avoid false negative (e.g., one cannot match what is not indexed since the match is exact). Larger indexes demand more storage and memory resources which leads to longer match processing times. Larger indexes are also more prone to false positive matches as they contain many duplicate or near duplicate copies of the content which in turn requires costly human review of false positive matches. This makes the current approaches impractical and too expensive in practice.

This method addresses this first challenge of false negative and index size with a new LSH fingerprint:
- using an LSH enables approximate matching instead of only exact matching,
- approximate matching can limit the index size by discarding near-duplicates that are within matching radius of existing index items and still avoid false negative.

2) The second challenge is content duplication. The inherent nature of public content is that it is copied, reused and remixed as enabled by the open availability of this content. In the specific case of software source code, there are thousand modified copies of popular software package like the Linux kernel. Duplication makes it difficult to pinpoint the correct content origin and can lead to many false positive.
This new method addresses this challenge these ways:

- By approximating content ancestry at indexing using pair-wise similarity to construct a phylogenetic-like tree will a) avoid overloading the index with near-duplicates that are still within matching radius of existing index items and still avoid false negative, and b) limit the size of the index limits the potential for false positives
- Furthermore, for code, the matching of whole code graphs and trees at once partitions and filters the search space to further avoid unrelated false positives. This can be used as a first filtering step in a discovery process.

Method description

The proposed method is realized through these steps:

Prepare content for indexing

In this step, the file content is converted to text and normalized. In the common case. text is used as input. In other cases, we extract text content of binary content such as image EXIT metadata or ELF string sections. In yet other cases, the content may be an actual tree, or graph of data, or a set of materialized and sorted paths, or a set of fingerprints or checksums.
When the input is text, the text is then normalized such as for instance converting to Unicode, converting text to lowercase, normalizing white spaces and punctuation.

The content is then split in tokens, aka. words. The split can be simply based on space or punctuation. Or can used advanced rules based on the natural language. Each word or token may further be processed such as applying stemming or lemmatization.

Another split could be achieved using a fixed length number of characters, or n-grams of characters or any other similar method suitable to split the text in tokens.
When the input is software source code, the code can be parsed in stream of code token using code lexing utilities, and optionally filtered (such as removing code comments), categorized (such as tagging literal string and number values, vs. programming language keywords vs. variables, classes and other programming constructs.)

When the input is not text, but existing fingerprints or checksums, this step may be bypassed.

The outcome of this step is a stream of normalized word tokens, possibly enriched with additional details such as a token type in a programming language, or a token position.

Group tokens in chunks and select chunks

In this step, the stream of tokens from the previous step is processed to group and select a subset of groups suitable to indexing. A common approach is to mode a sliding window of a given size W over the stream of tokens and use a heuristic to select such a sliding window as significant.
The length of W can vary typically from a value of W=8 to about W=32, or larger values.
For character-based ngrams, values of 24 to 32 are typical with shorter values yielding better recall but lower precision. The heuristic can be based on hashing each token content in a window and selecting the windows with certain properties. For instance, [AIKEN] winnowing heuristic that uses a second sliding window over ngrams and selects the ngram hash with the minimum value in each window. Or [ABDEL-HAMID] that compares winnowing with other methods, introduces hailstorm and describes other approaches. Or [SEO] that presents various methods such as hash breaking or Discrete Cosine Transform.

When the input is not text, but existing fingerprints or checksums, this step may be bypassed.

In all cases, the outcome of this step is a stream of token groups, possibly enriched with additional details such as its origin, its position and length.

Compute approximate fingerprints

In this step, each group of tokens from the previous step is processed to compute a fingerprint. This fingerprint is computed based on either of the sequences:
- word ngrams of length N (where N << W)
- each words taken individually
- all the words in a window

The preferred approach is to use short ngrams, with N typically between 3 and 5.

The computed fingerprint can be based on random projections, min-wise independent permutations or similar dimensionality reduction fingerprints.

When the input is not text, but existing fingerprints or checksums, this step may be be performed using a dedicated procedure to combine these fingerprints or checksums to create a new fingerprint.

In all cases, the outcome of this step is a stream of token groups, possibly enriched with additional details such as its origin, its position and length.

Determine content precedence

In this step, each fingerprint from the previous step is pre-processed to be stored and indexed.
The method is to query the existing index for a fingerprint, and if there is a match within a certain similarity threshold then this new fingerprint may not be stored as-is, and the existing fingerprint may be instead related to the newly indexed content origin.

If the existing content is determined to originate from a less reputable origin that the new content, then the new content would become the new reference origin for this fingerprint and the old content a secondary origin.

The same approach can be performed taking into account the available content publication timeline. The oldest published content is assumed to be a better origin for the content that an origin that was published later. The later origin should become secondary and the earlier origin should become the primary.

Alternatively, or in addition, a preferred content origin from a pair of similar content may be determined with this method: we compute the content containment between the whole content matched origin and the whole content of the candidate for indexing. When the candidate indexed content that represents the highest proportion of an origin in the pair of origins, that origin is deemed to be a better origin for that content that an origin that contains only a smaller fraction of that content. This method can be further enhanced by taking into account the content publication timeline either first or in conjunction.

The outcome of this step is a mapping of fingerprints to index and their origin; and set of updates to existing, already indexed origins and fingerprints.

Store and index content

In this step, each selected fingerprint from the previous step is stored and indexed. As needed, primary origins are updated according to the precedence we determined in the previous step.

The storage can be using plain sorted files, indexed files, or a proper database such as relational database, a key-value store or a vector database, when the fingerprint effectively forms a feature vector.
The preferred storage and index is such that the fingerprints can be efficiently queried for similarity using a similarity threshold and metric, such as a cosine similarity between vectors, or the hamming distance between their binary representation.
A possible and common implementation for hamming distance is described consist in breaking a fingerprint in equal-sized parts, for instance 4 parts of 32 bits each with a 128 bits fingerprint, and index each of these separately for lookup. If any of the part is found to match, the results are guaranteed to be with a 4-1=3 hamming distance, as detailed in [CHARIKAR].
Another implementation may use a vector database with builtin nearest neighbor query capabilities.
It is typically useful to track and store not only the origin of a content fingerprint, but also its original length and start position for use in the next query step.

The outcome of this step is database and index of origins and fingerprints ready for querying.

Query for similarity

In this step, new content is processed to find its similarity with pre-existing, index content.

The new content is pre-processed as if it were to be indexed and the database and index is queried using a procedure similar to the "index precedence" step.
When multiple matching fingerprints are found within a certain similarity threshold to come from the same original indexed content, these matches are sorted by the original position of the selected fingerprints, and merged based on the overlap between matched fingerprints determined based on the combination of their start positions and length.
This procedure ensures that longer matches are assembled from contiguous matched fragments.
A secondary procedure can optionally perform a comprehensive hamming distance computation between each pair of original and matched content fingerprints.
Yet another secondary procedure can optionally perform a comprehensive sequence alignment between the original and the matched content, either directly using the original index and queried content, or indirectly using fingerprints as proxy for the content.

Eventually, the returned results are ranked based on the size of the matches and the proportion of an original content that was matched, possibly across multiple content items.

Fingerprint variants

A fingerprint variant is designed using random projections [JOHNSON-LINDENSTRAUSS], but without taking into account global factors like the TF/IDF from [CHARIKAR].
In this fingerprint scheme, an alternative is to keep more than a single bit, and accumulate intermediate computation results for each bit. With this alternative variant, it becomes possible to more efficiently recombine multiple fingerprints together as the bit averaging becomes more efficient when intermediate results are kept.
The base variant otherwise consist of applying a progressive hierarchical matching approach, where the fingerprint is split in non-overlapping fragments and a fingerprint fragment is first considered for exact matching, and then additional fragments are considered either only for the subset of matched origins or across all origins, and the size of the fragments determines the precision and recall of the approximate search.
In a variant of this progressive matching approach a first matching procedure may use a short fingerprint fragment and then perform a secondary match on a longer fragment to first partition the search space and then perform a more costly search procedure on the subset of the search space.

Applications

These are examples of applications of the proposed method with AI-Generated Code Search, and with Software Origin Discovery for software supply chain security. The methods described here have other applications beyond these including training data content deduplication for AI and LLMs.

Application to AI-generated code search

Generative AI engines and Large Language Models (LLMs) are emerging as viable tools for software developers to automate writing code. These engines and LLMs are trained on publicly available, free and open source (FOSS) code.

AI-generated code can inherit the license and vulnerabilities of the FOSS code used for its training. It is essential and urgent to identify AI-generated source code, as it threatens the foundation of open source development and software development and raises major ethical, legal and security questions.

This new approach can be used to identify and detect if AI-generated code is derived from existing FOSS apply these processes to code fragments approximate similarity search and identify if source code is AI-generated and report which FOSS project it derives from.

Approximate code fragments matching on smaller indexes is required to accommodate the variety of AI-generated code and the growing volume of open source code used to train the backing LLMs.

This new approach to code matching with locality-sensitive hashing (LSH) using random projections tunable for precision and recall will avoid an ever increasing code index size. This will make it practical for users to identify AI-generated code at scale.

Providing a practical solution to discover AI-generated code fragments will improve the trustworthiness of both AI-generated and non-generated code - and open source code in particular - by reporting its AI-generated status and ultimate FOSS origin.

Enabling trustworthy, safer and efficient usage of LLM-based code generation with this critical knowledge will allow improving overall programmer productivity and adoption of responsible AI-code generation tools.

Application to Software Supply Chain Integrity and Security

The proposed approach will further open source software supply chain security by enabling efficient and accurate open source software code origin determination at scale.

Free and open source software (FOSS) is the foundation of all modern software. Software developers can provision FOSS packages easily using modern package managers, but they often loose track of which exact package they are using. FOSS packages also form deep dependency graphs where a package depends on another package and it is hard to keep an inventory of all reused code.

Yet it is imperative to know the origin of reused FOSS to find known security issues, bugs and licensing terms; and to secure a software supply chain and create an accurate SBOM (Software Bill of Material) of all reused code, now a federal government requirement with the US Executive Order 14028 on "Improving the Nation’s Cybersecurity", NIS2, and similar requirements in Europe with the Cyber Resilience Act (CRA), PLD and DORA regulations.

There are multiple techniques to determine which FOSS code is reused and embedded in a system or application:
- Tracing the software build to keep track the software packages,

- Parsing package manifests files to extract their metadata,
- Finding similar code by matching against a large existing index of open source.

The first two approaches are fairly straightforward and the basis of most available solutions today.
The third approach based on code similarity matching has not been deployed successfully because of issues of index size, false positive (finding the wrong code) or false negative (failing to find code that is similar even though it exists) and the high human review efforts required to make this work at scale.

This approach will match code based on similarity:
- With a tunable precision and recall search,
- Will return the correct origin among the large number of existing FOSS package copies and near duplicates,
- And will be doing this using fewer storage and compute resources that the state of the art solutions.

The key to this method is to match code approximately with a new locality sensitive hashing and its binary code serialization (aka. fingerprint) with these properties:

- Supporting multiple precision levels in a single fingerprint: existing fingerprint schemes have a set size that defines the amount of information and the precision of the matching.
- Working correctly on smaller inputs: existing fingerprint schemes work poorly or not at all on smaller inputs
- Recombining multiple fingerprints in a new fingerprint: this is a desirable property that is available only in some schemes today such as min-wise independent permutations.
- Using smaller, fixed bit length: several fingerprint schemes require fairly long bit strings (like 1000 bits or more) to work efficiently`
- Not depending on global, index-wide constants: several LSH include index-wide terms in the computation such as a the index work frequency (for a tf/idf). These are terms that are almost constant but require computation before an LSH can be computed. This makes these solutions impractical to deploy as the volume of FOSS code is evolving continuously and it is impossible to recompute all the LSH on each index change.
- Scoring of matches using a hamming distance between bit strings.

This fingerprinting scheme is computed whole code trees structure and content, code file content or AST (abstract syntax tree), symbols, and code chunks (aka. snippets) using a content-defined chunking algorithm.

When used at indexing time, the system does self match to infer a code phylogenetic tree to approximate the code ancestry and automatically determine the correct, ancestral code origin.

References

- [AIKEN] Winnowing: Local Algorithms for Document Fingerprinting
https://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf

- [ABDEL-HAMID] Detecting the origin of text segments efficiently
https://infoscience.epfl.ch/record/138562/files/OriginTextSegments.pdf

- [SEO] Local Text Reuse Detection
https://ciir-publications.cs.umass.edu/pub/web/getpdf.php?id=812

- [JOHNSON-LINDENSTRAUSS] Johnson-Lindenstrauss lemma
https://en.wikipedia.org/wiki/Johnson–Lindenstrauss_lemma

- [BRODER] On the resemblance and containment of documents
https://web.archive.org/web/20150131043133/http://gatekeeper.dec.com/ftp/pub/dec/SRC/publications/broder/positano-final-wpnums.pdf

- [CHARIKAR] Similarity Estimation Techniques from Rounding Algorithms
https://www.cs.princeton.edu/courses/archive/spr04/cos598B/bib/CharikarEstim.pdf

- [HENZIGER] Finding near-duplicate web pages: A large-scale evaluation of algorithms
https://infoscience.epfl.ch/entities/publication/ca590109-0048-454b-89d9-4cbb9b64b3bc

- [BURROWS] Efficient plagiarism detection for large code repositories
https://people.eng.unimelb.edu.au/jzobel/fulltext/spe07.pdf

- [SOOD-LOGUINOV] Probabilistic near-duplicate detection using simhash
https://irl.cse.tamu.edu/people/sadhan/papers/cikm2011.pdf

- [STEIN] Efficient plagiarism detection for large code repositories
https://downloads.webis.de/publications/papers/stein_2005a.pdf

- [LULU] Overview of fingerprinting methods for local text reuse detection
https://www.researchgate.net/profile/Boumediene-Belkhouche/publication/310441711_Overview_of_Fingerprinting_Methods_for_Local_Text_Reuse_Detection/links/5bbc8344299bf1049b783ce0/Overview-of-Fingerprinting-Methods-for-Local-Text-Reuse-Detection.pdf

- [JEKABSONS] Evaluation of Fingerprint Selection Algorithms for Local Text Reuse Detection
http://www.cs.rtu.lv/jekabsons/Files/Jek_ACSS2020.pdf