Chapter 4: Shared Code Analysis

DATA SCIENCE IN SECURITY

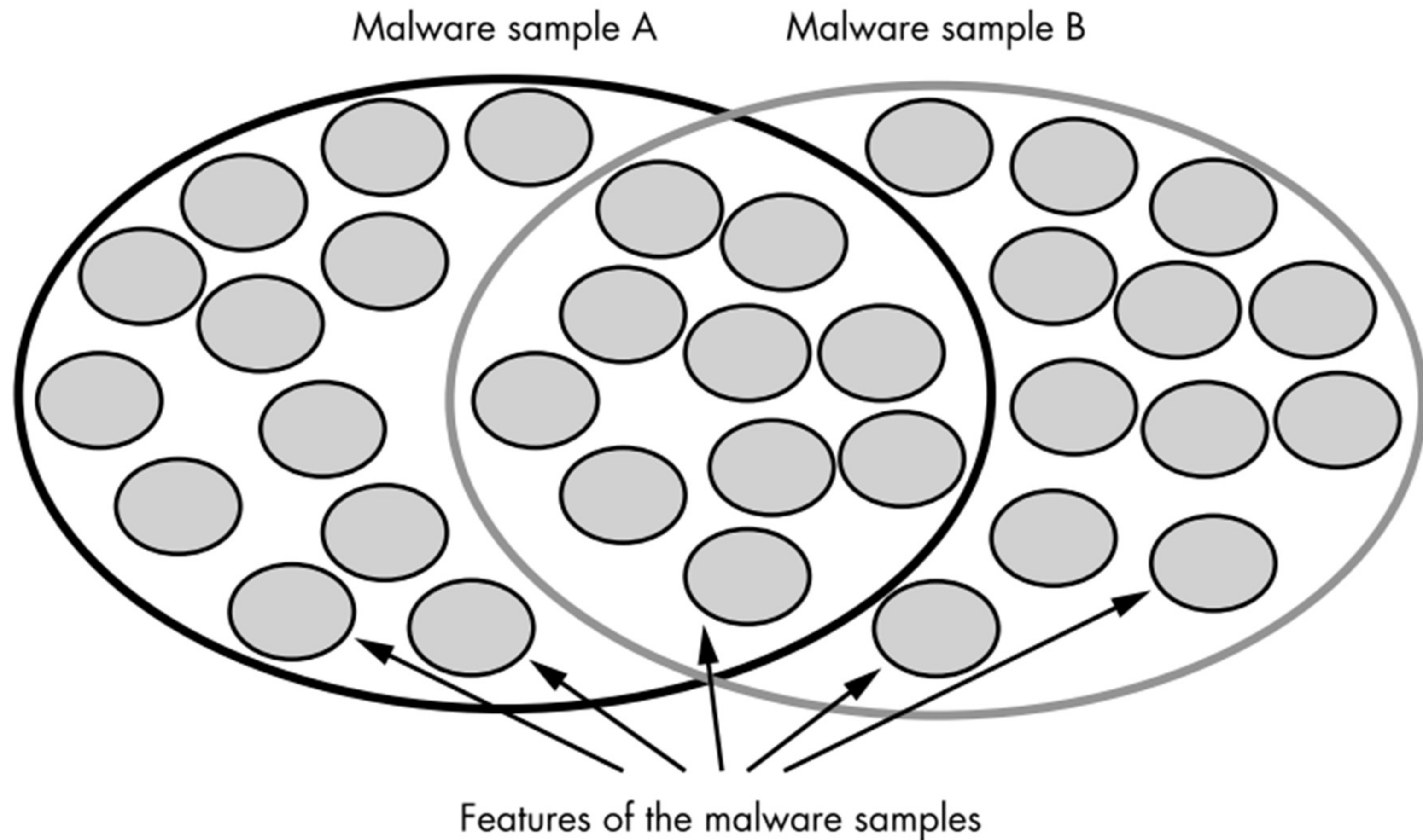
- Shared code analysis
  - also called similarity analysis
  - the process by which we compare two malware samples by estimating the percentage of pre-compilation source code they share
  - differs from shared attribute analysis
  - helps identify samples that can be analyzed together
    - they
      - were generated from the same malware toolkit
      - or are different versions of the same malware family
  - can determine whether the same developers could have been responsible for a group of malware samples

- Shared code analysis

```
Showing samples similar to WEBC2-GREENCAT_sample_E54CE5F0112C9FDFE86DB17E85A5E2C5
Sample name                                                      Shared code
[*] WEBC2-GREENCAT_sample_55FB1409170C91740359D1D96364F17B       0.9921875
[*] GREENCAT_sample_55FB1409170C91740359D1D96364F17B             0.9921875
[*] WEBC2-GREENCAT_sample_E83F60FB0E0396EA309FAF0AED64E53F       0.984375
    [comment] This sample was determined to definitely have come from the advanced persistent
threat group observed last July on our West Coast network
[*] GREENCAT_sample_E83F60FB0E0396EA309FAF0AED64E53F             0.984375
```

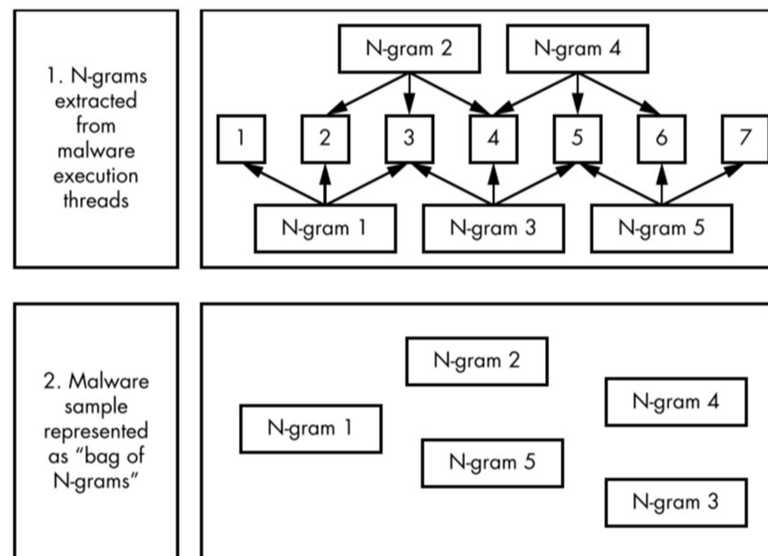
- we group malware samples into “bags of features” before comparing
  - Feature: any malware attribute we might possibly want to consider
    - E.g. the printable strings we can extract
  - we think of malware as a bag of independent features for mathematical convenience

preparing Samples for comparison by extracting Features



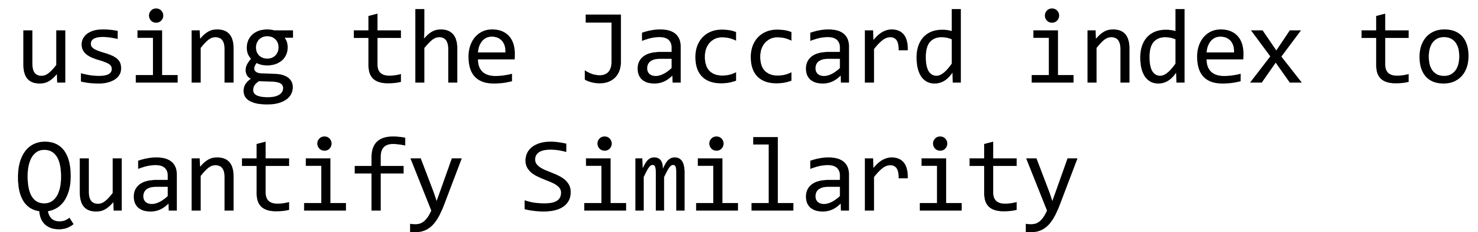
preparing Samples for comparison by extracting Features

- What are N-Grams?
  - a subsequence of events that has a certain length, N, of some larger sequence of events
  - Can be extracted by sliding a window over the sequential data



preparing Samples for comparison by extracting Features

- What are N-Grams?
  - In malware analysis
    - we would extract N-grams of sequential malware API calls
    - Then we would represent the malware as a bag of N-grans
    - incorporates sequence information into features comparison
      - Good, when order matters in the comparison
        - malware calls A before B, which was observed before calling C
      - Bad, when order is superfluous
        - malware randomizing the order of API calls A, B, and C on every run

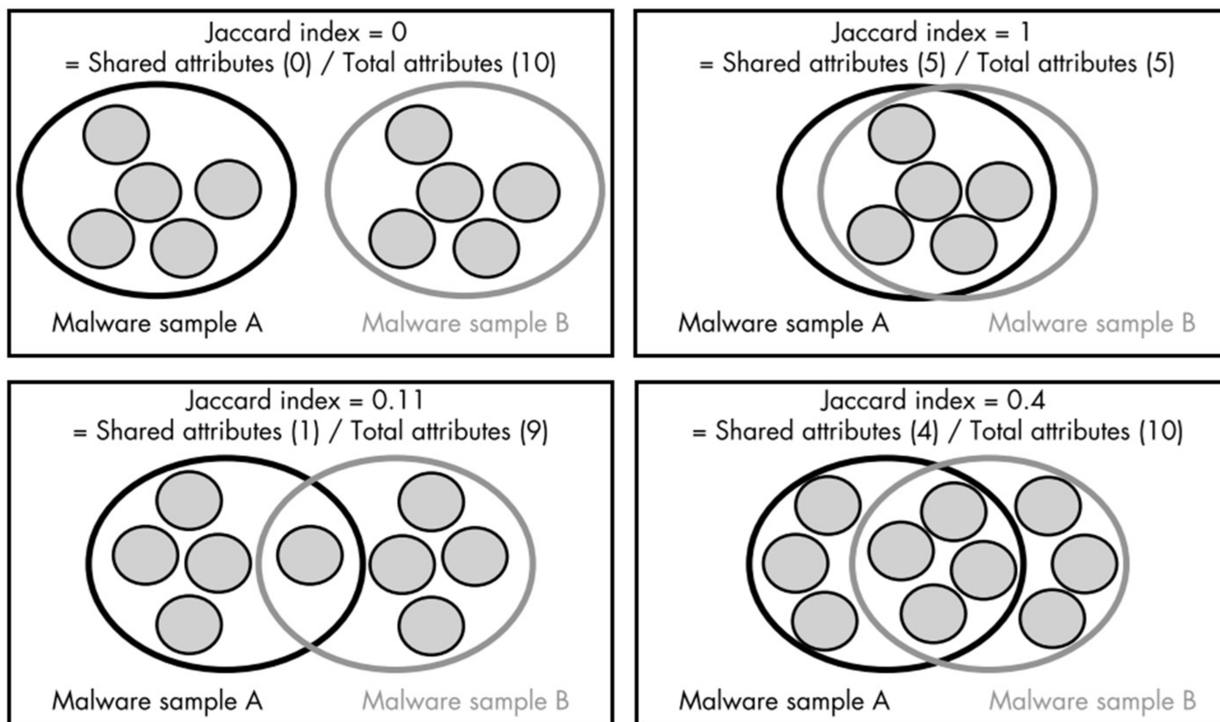


using the Jaccard index to Quantify Similarity

- We need a similarity function that should have the following properties
  - It yields a normalized value
  - help us make accurate estimates of code sharing between two samples
  - should be easily understandable why the function models code similarity well

using the Jaccard index to Quantify Similarity

- The Jaccard index
  - has all these properties
  - emerged as the most widely adopted

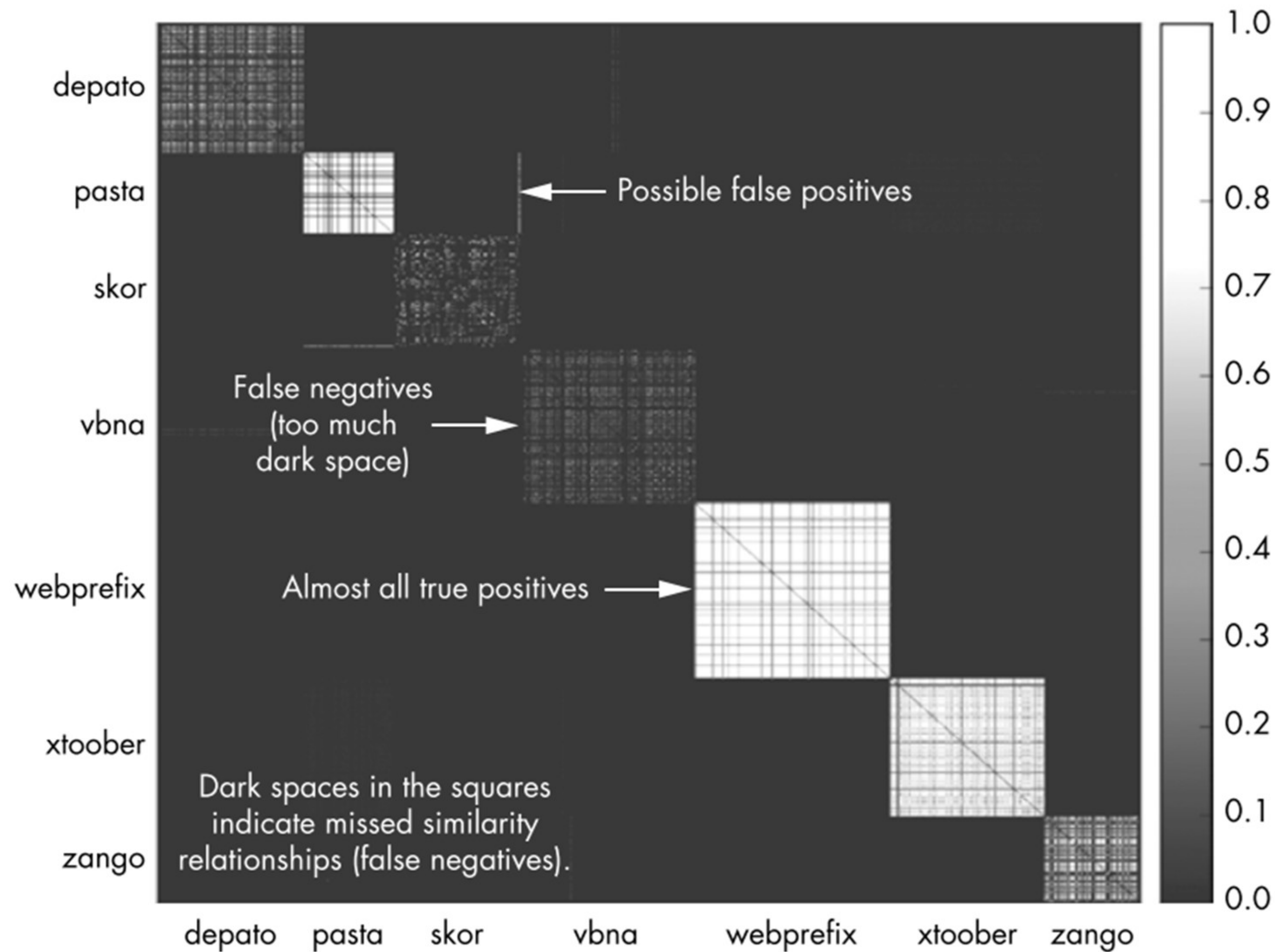


- Consider four similarity feature:
  - instruction sequence-based similarity
  - Strings based similarity
  - Import Address Table–based similarity
  - Dynamic API call–based similarity
- To compare above features
  - we'll use a similarity matrix visualization technique

using Similarity Matrices to evaluate Malware Shared code estimation Methods

	Sample 1	Sample 2	Sample 3	Sample 4
Sample 1	Similarity between 1 and 1	Similarity between 1 and 2	Similarity between 1 and 3	Similarity between 1 and 4
Sample 2	Similarity between 2 and 1	Similarity between 2 and 2	Similarity between 2 and 3	Similarity between 2 and 4
Sample 3	Similarity between 3 and 1	Similarity between 3 and 2	Similarity between 3 and 3	Similarity between 3 and 4
Sample 4	Similarity between 4 and 1	Similarity between 4 and 2	Similarity between 4 and 3	Similarity between 4 and 4

using Similarity Matrices to evaluate Malware Shared code estimation Methods



Instruction Sequence-Based Similarity

- most intuitive way to compare two malware binaries
- requires disassembling malware samples using
  - E.g. the linear disassembly
- we can use the N-gram approach
  - Value of N depends on our analysis goals.
    - The larger N
      - harder it will be for malware samples' sequences to match.
      - helps identify only samples that are highly likely to share code
    - The smaller N
      - looks for subtle similarities between samples
      - Can be used if you suspect that the samples employ instruction reordering

Instruction Sequence-Based Similarity

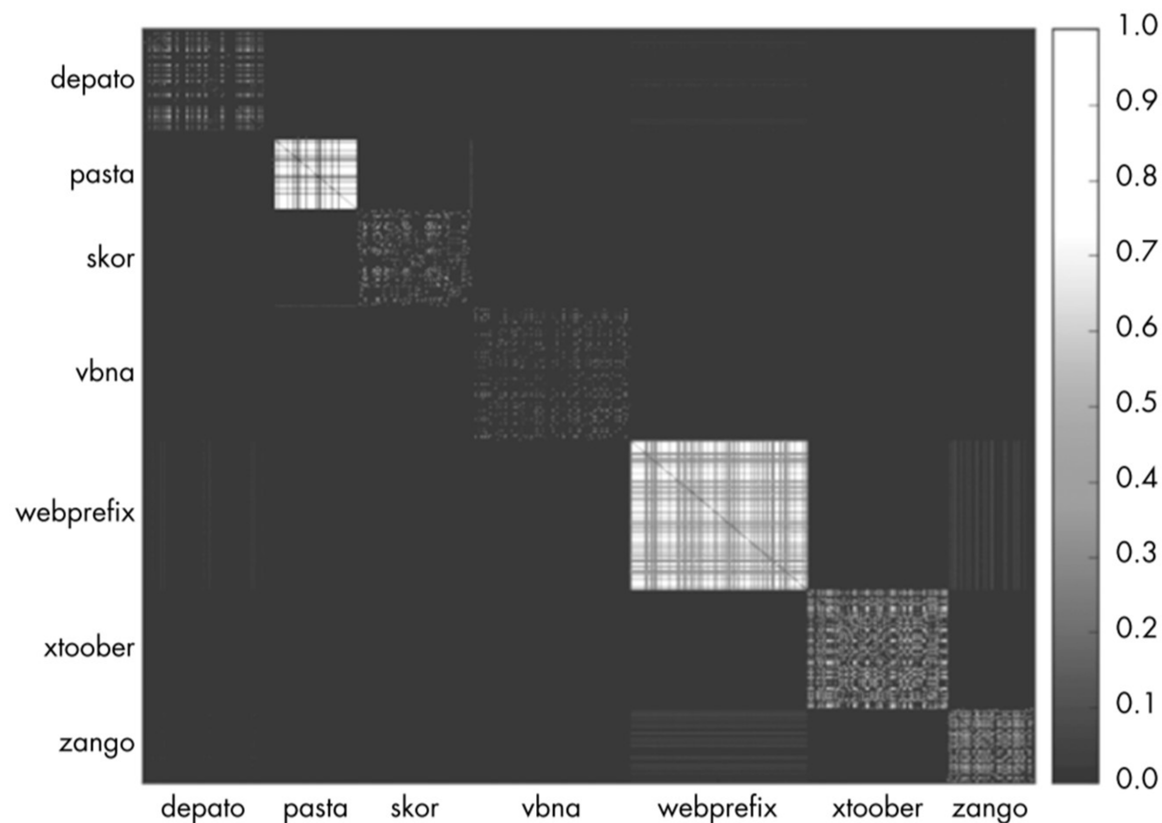


Figure 5-7: The similarity matrix generated using instruction N-gram features. Using N = 5, we completely miss many families' similarity relationships but do well on webprefix and pasta.

Instruction Sequence-Based Similarity

- Advantage: few false positives
- Disadvantage: can miss many code-sharing relationships
  - because malware samples may be packed
  - Even when we unpack our malware samples:

```
int f(void) {
    int a = 1;
    int b = 2;
  ❶ return (a*b)+3;
}
```

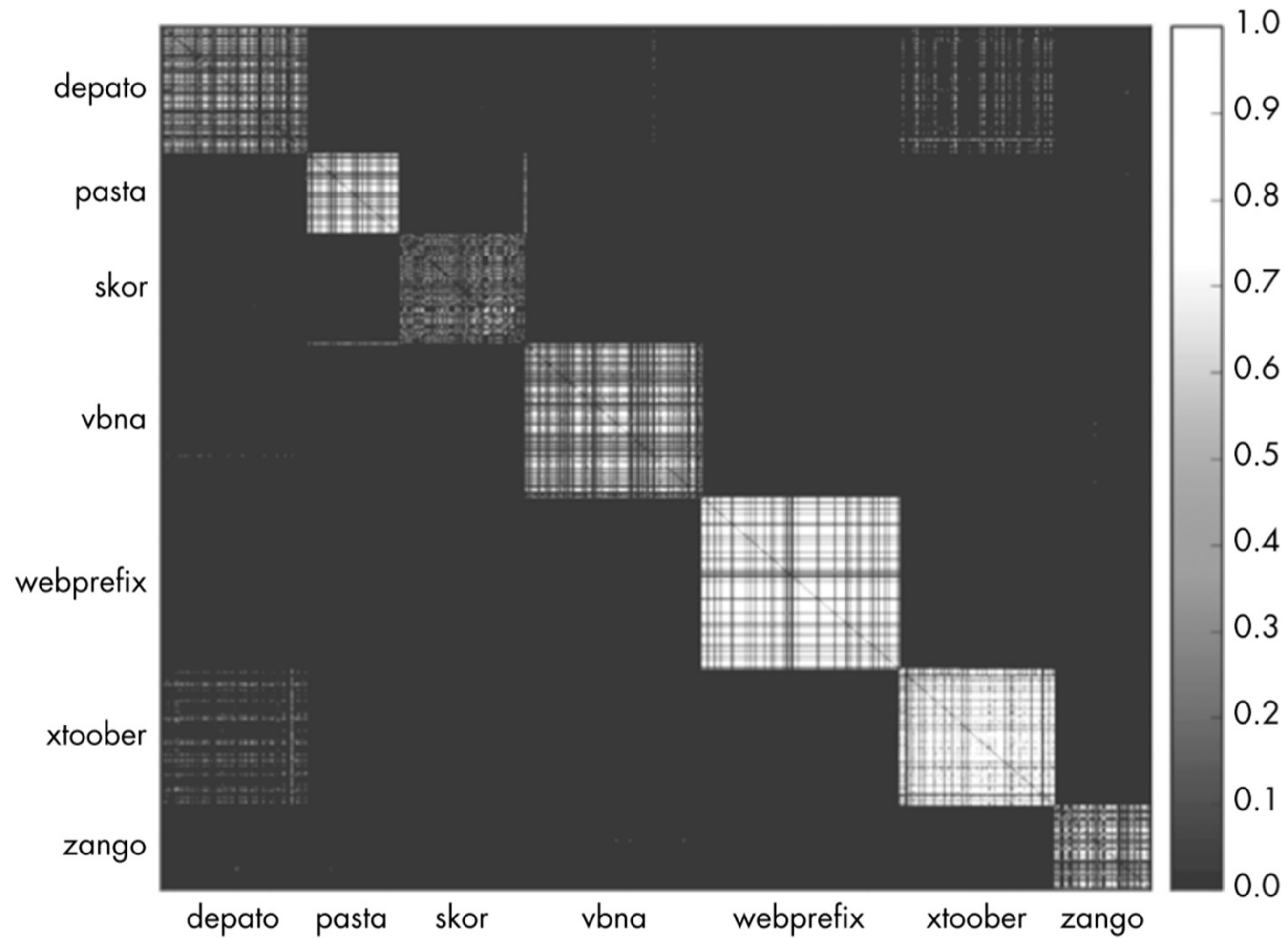
```
movl    $1, -4(%rbp)
movl    $2, -8(%rbp)
movl    -4(%rbp), %eax
imull   -8(%rbp), %eax
addl    $3, %eax
```

```
movl    $5, %eax
```

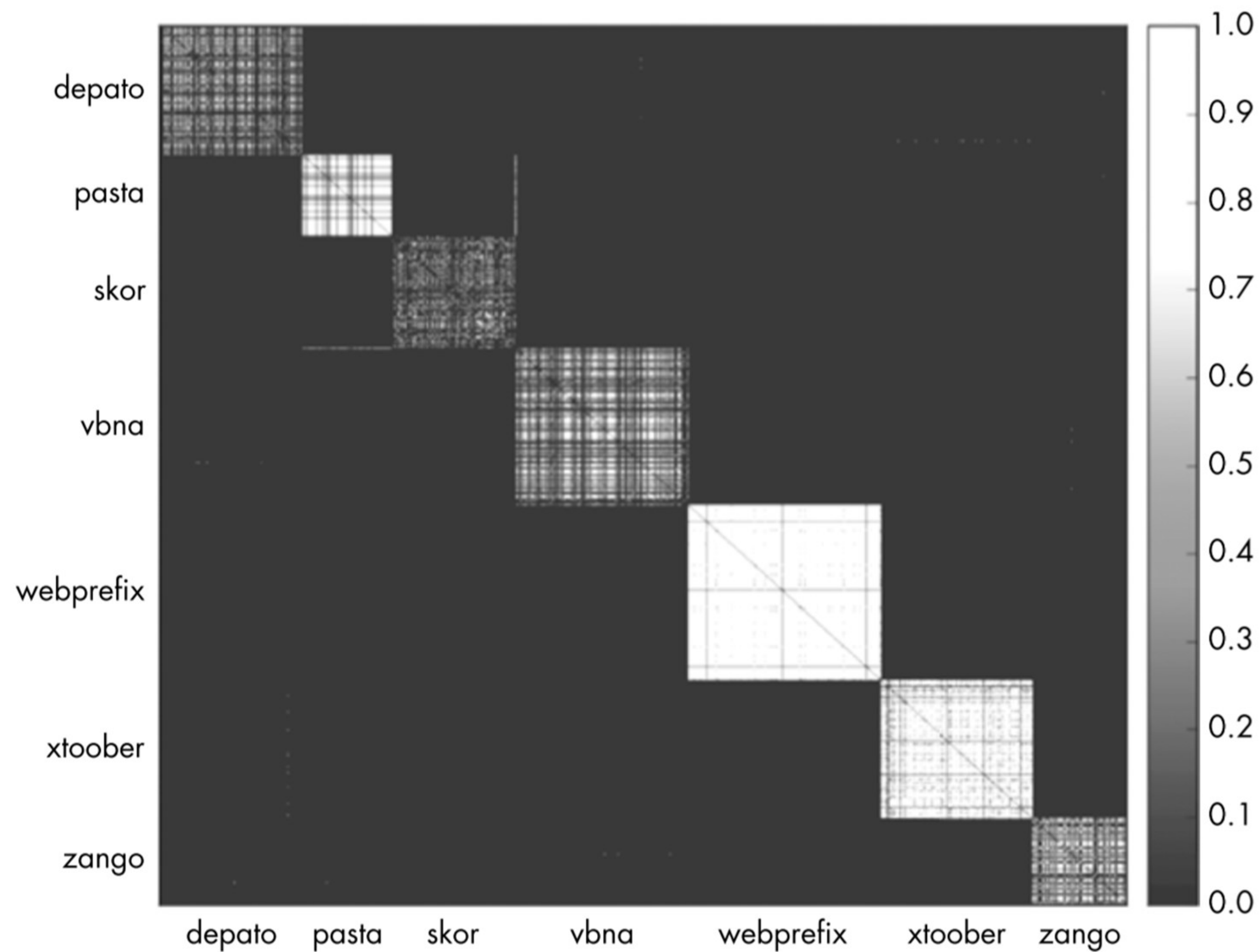
  - many malwares are authored in languages like C#
    - contain standard assembly code that interprets the higher-level languages' bytecode
    - share very similar x86 instructions
    - their actual bytecode come from very different source code

- can be computed by
  - extracting all contiguous printable sequences of characters in the samples
  - computing the Jaccard index between all pairs of malware samples based on their shared string relationships
- Can gets around the compiler problem
  - compilers do not transform strings in a binary

Strings-Based Similarity

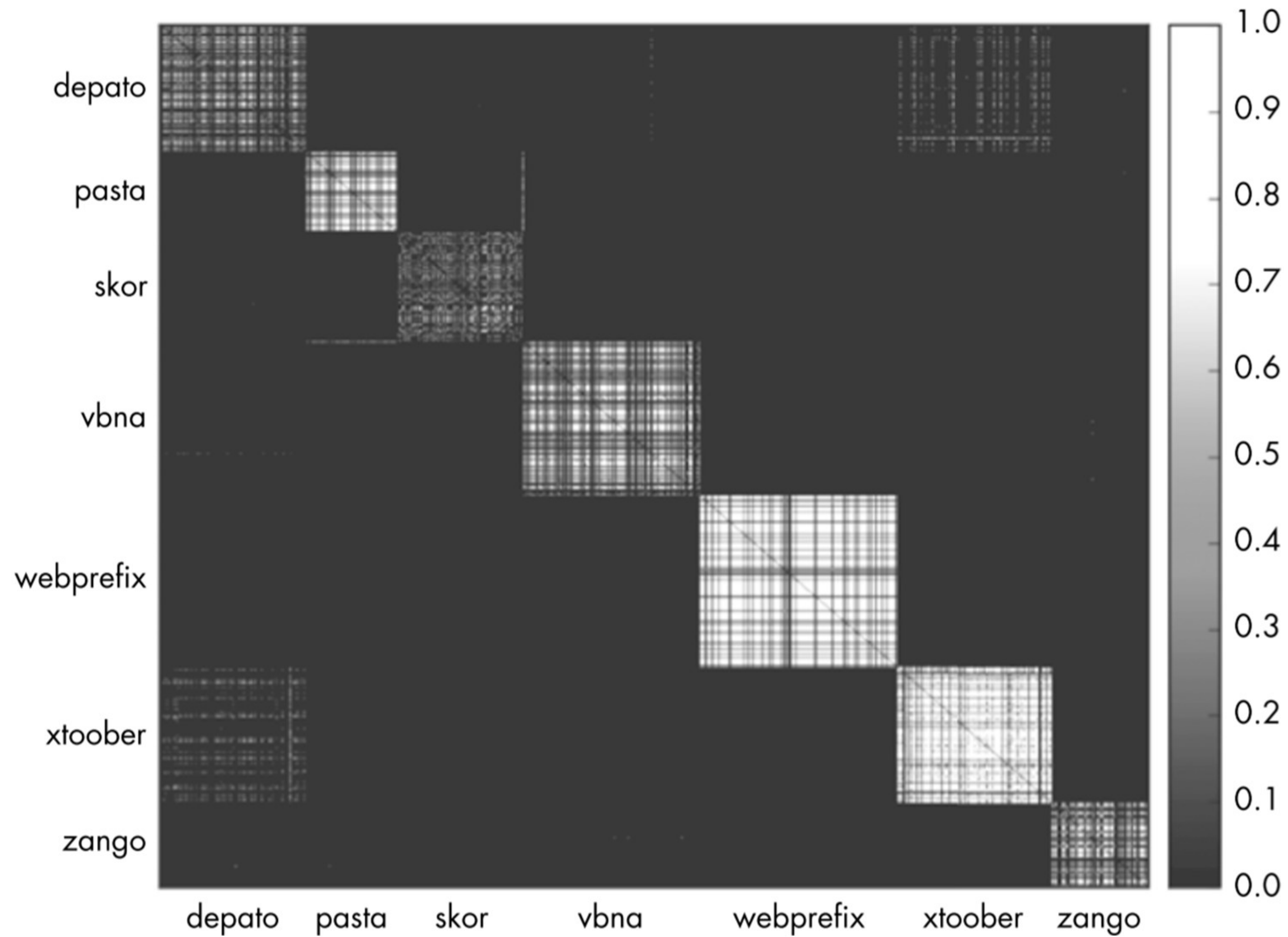


Import Address Table–Based Similarity

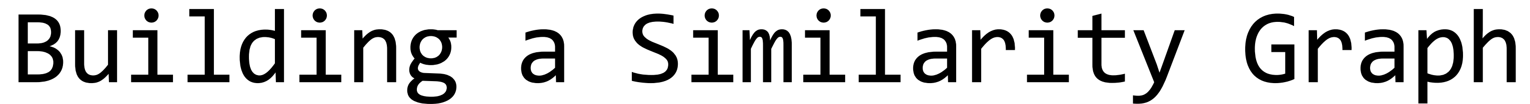


- To implement this approach, you'll need to
  - run malware samples in a sandbox
  - record the API calls they make
  - extract N-grams of API calls from the dynamic logs
  - finally compare the samples by taking the Jaccard index between their bags of N-grams.

Dynamic API Call-Based Similarity



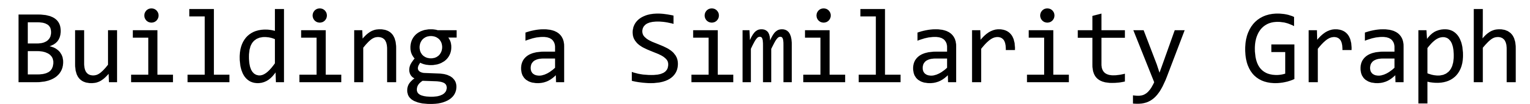
- The imperfect results here show
  - Simply running malware in a sandbox is not sufficient to trigger many of its behaviors.
  - some samples detect that they're running in a sandbox and then promptly exit execution
- In summary
  - dynamic API call sequence similarity isn't perfect
  - but it can provide impressive insight into similarities between samples.




```
#!/usr/bin/python

import argparse
import os
import networkx
from networkx.drawing.nx_pydot import write_dot
import itertools

def jaccard(set1, set2):
    """
    Compute the Jaccard distance between two sets by taking
    their intersection, union and then dividing the number
    of elements in the intersection by the number of elements
    in their union.
    """
    intersection = set1.intersection(set2)
    intersection_length = float(len(intersection))
    union = set1.union(set2)
    union_length = float(len(union))
    return intersection_length / union_length
```



Building a Similarity Graph

```
def getstrings(fullpath):
    """
    Extract strings from the binary indicated by the 'fullpath'
    parameter, and then return the set of unique strings in
    the binary.
    """
    strings = os.popen("strings '{0}'".format(fullpath)).read()
    strings = set(strings.split("\n"))
    return strings

def pecheck(fullpath):
    """
    Do a cursory sanity check to make sure 'fullpath' is
    a Windows PE executable (PE executables start with the
    two bytes 'MZ')
    """
    return open(fullpath).read(2) == "MZ"
```

Building a Similarity Graph

```
If __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Identify similarities between malware samples and build similarity graph"
    )

    parser.add_argument(
        "target_directory",
        help="Directory containing malware"
    )

    parser.add_argument(
        "output_dot_file",
        help="Where to save the output graph DOT file"
    )

    parser.add_argument(
        "--jaccard_index_threshold", "-j", dest="threshold", type=float,
        default=0.8, help="Threshold above which to create an 'edge' between samples"
    )

    args = parser.parse_args()
```

Building a Similarity Graph

```
If __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Identify similarities between malware samples and build similarity graph"
    )

    parser.add_argument(
        "target_directory",
        help="Directory containing malware"
    )

    parser.add_argument(
        "output_dot_file",
        help="Where to save the output graph DOT file"
    )

    parser.add_argument(
        "--jaccard_index_threshold", "-j", dest="threshold", type=float,
        default=0.8, help="Threshold above which to create an 'edge' between samples"
    )

    args = parser.parse_args()
```

Building a Similarity Graph

```
malware_paths = []  # where we'll store the malware file paths
malware_features = dict()  # where we'll store the malware strings
graph = networkx.Graph()  # the similarity graph

for root, dirs, paths in os.walk(args.target_directory):
    # walk the target directory tree and store all of the file paths
    for path in paths:
        full_path = os.path.join(root, path)
        malware_paths.append(full_path)

# filter out any paths that aren't PE files
malware_paths = filter(pecheck, malware_paths)

# get and store the strings for all of the malware PE files
for path in malware_paths:
    features = getstrings(path)
    print "Extracted {0} features from {1} ...".format(len(features), path)
    malware_features[path] = features

    # add each malware file to the graph
    graph.add_node(path, label=os.path.split(path)[-1][:10])
```

Building a Similarity Graph

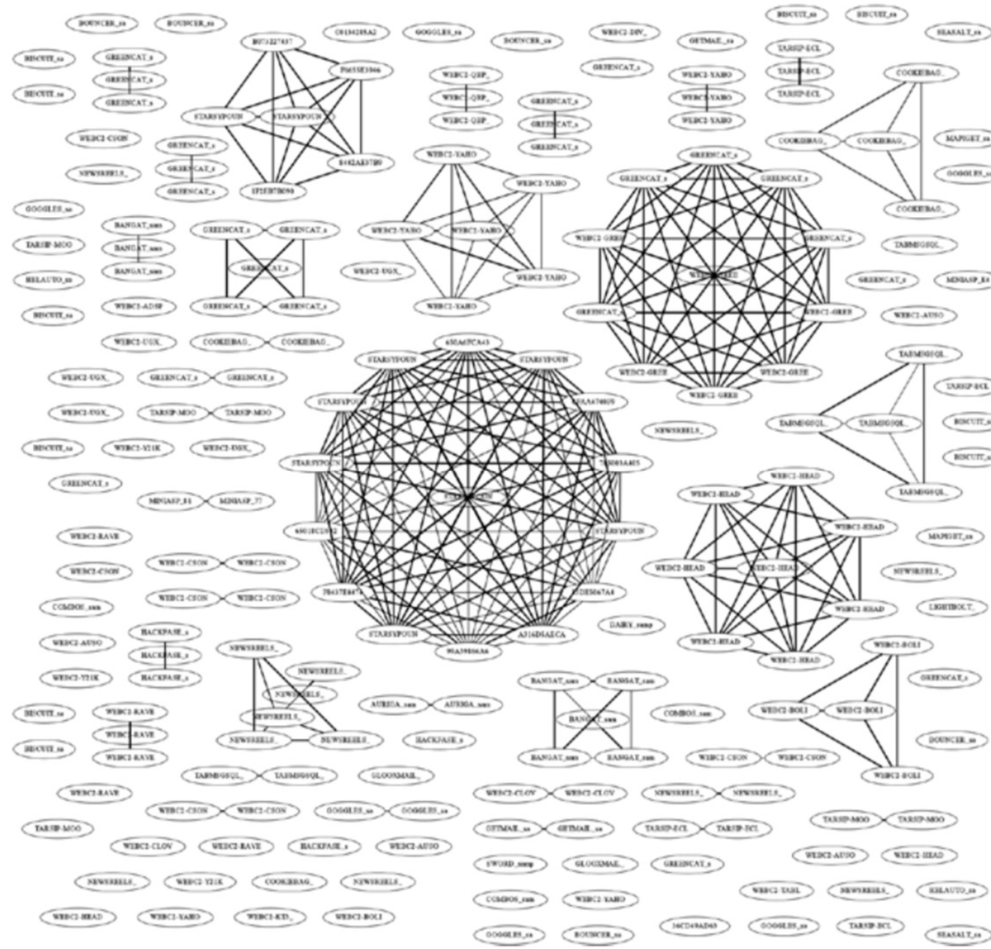
```
# iterate through all pairs of malware
for malware1, malware2 in itertools.combinations(malware_paths, 2):

    # compute the jaccard distance for the current pair
    jaccard_index = jaccard(malware_features[malware1], malware_features[malware2])

    # if the jaccard distance is above the threshold, add an edge
    if jaccard_index > args.threshold:
        print malware1, malware2, jaccard_index
        graph.add_edge(malware1, malware2, penwidth=1+(jaccard_index-args.threshold)*10)

# write the graph to disk so we can visualize it
write_dot(graph, args.output_dot_file)
```

Building a Similarity Graph

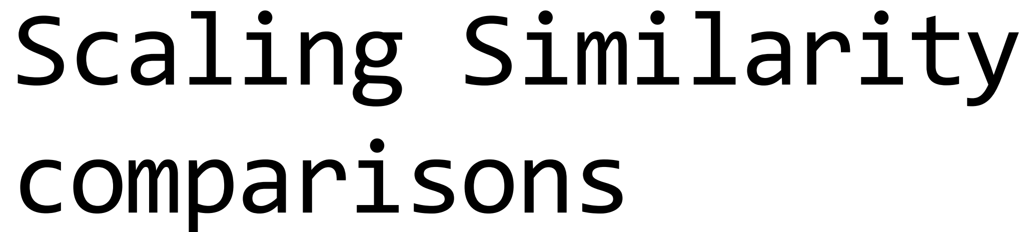


Scaling Similarity comparisons

- Previous codes doesn’t work well for a large number of malware samples
  - number of necessary Jaccard index computations

$$\frac{n^2 - n}{2}$$

  - A dataset that has 50,000 samples would require 1,249,975,000 Jaccard index computations!



Scaling Similarity comparisons

- we need to use randomized comparison approximation algorithms
  - allow for some error in our computation of comparisons in exchange for a reduction in computation time.
  - minhash serves this purpose for us beautifully.
    - allows us to compute the Jaccard index using approximation
      - avoid computing similarities between nonsimilar malware samples below some predefined similarity threshold
    - so that we can analyze shared code relationships between millions of samples

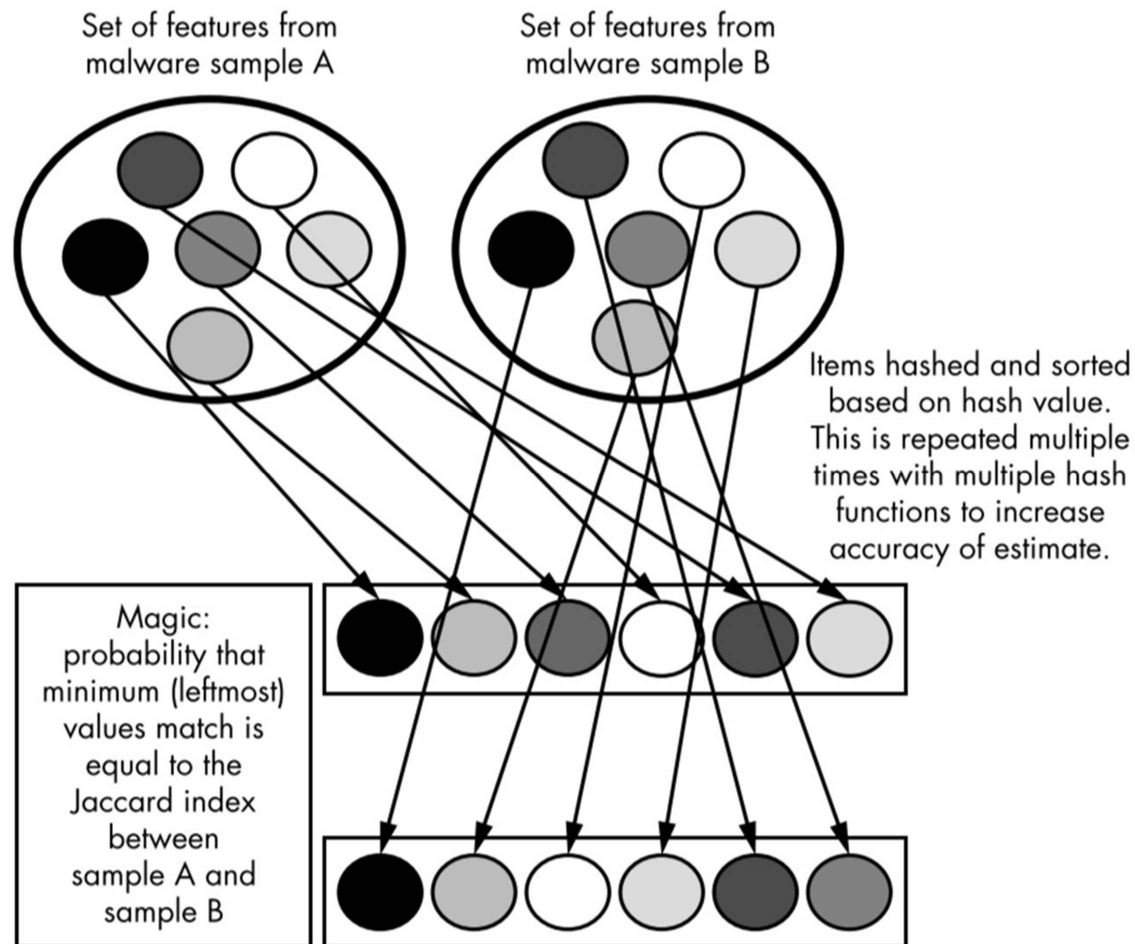
Minhash in a Nutshell

- Minhash takes a malware sample's features
  - hashes them with k hash functions.
  - For each hash function
    - we retain only the minimum value of the hashes computed over all the features
  - the set of malware features is reduced to a fixed size array of k integers
    - which we call the minhashes.

Minhash in a Nutshell

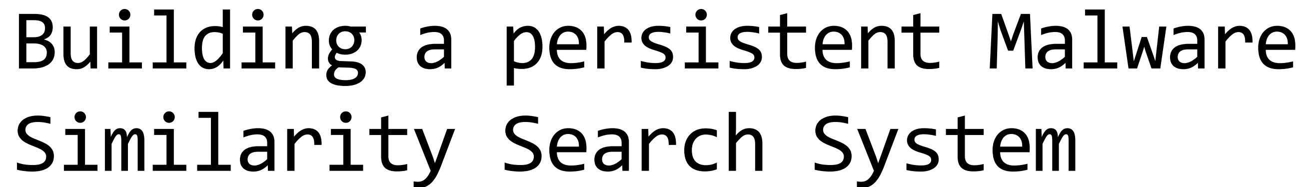
- To compute the approximate Jaccard index
  - you now just need to check how many of the k minhashes match, and divide that by k
  - Magically, the obtained number is a close approximation of the true Jaccard index between any two samples.
- The benefit of using minhash
  - it’s much faster to compute.
  - we can even use minhash to index malware in a database
    - we only need to compute comparisons between malware samples that at least one of their hashes matched

Minhash in a Nutshell



Minhash in a Nutshell

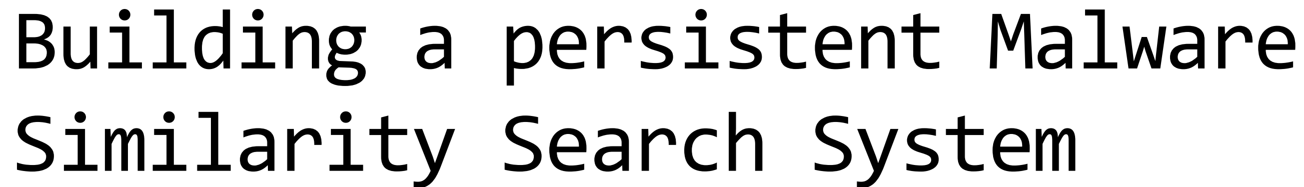
- using minhashes to speed up the search
  - The standard approach:
    - use sketching combined with database indexing
    - we compare only samples that we already know are highly likely to be similar.
    - a sketch is made by hashing multiple minhashes together.
    - When we get a new sample
      - we find any sketches that match the new sample's sketches.
      - the new sample is compared with the matching samples using their minhash arrays to approximate the Jaccard index




```
#!/usr/bin/python

import argparse
import os
import murmur
import shelve
import numpy as np
from listings_5_2_to_5_6 import *

NUM_MINHASHES = 256
SKETCH_RATIO = 8
```




```
def wipe_database():
    """
    This problem uses the python standard library 'shelve' database to persist
    information, storing the database in the file 'samples.db' in the same
    directory as the actual Python script. 'wipe_database' deletes this file
    effectively reseting the system.
    """
    dbpath = "/".join(__file__.split('/')[:-1] + ['samples.db'])
    os.system("rm -f {0}".format(dbpath))

def get_database():
    """
    Helper function to retrieve the 'shelve' database, which is a simple
    key value store.
    """
    dbpath = "/".join(__file__.split('/')[:-1] + ['samples.db'])
    return shelve.open(dbpath,protocol=2,writeback=True)
```

Building a persistent Malware Similarity Search System

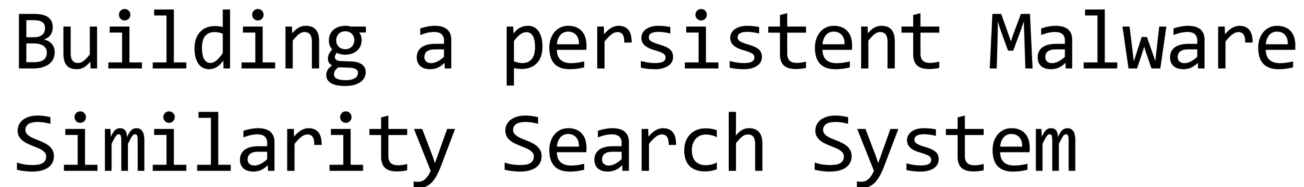
```
def minhash(features):
    """
    This is where the minhash magic happens, computing both the minhashes of
    a sample's features and the sketches of those minhashes. The number of
    minhashes and sketches computed is controlled by the NUM_MINHASHES and
    NUM_SKETCHES global variables declared at the top of the script.
    """
    minhashes = []
    sketches = []
 ❶ for i in range(NUM_MINHASHES):
        minhashes.append(
          ❷ min([murmur.string_hash(`feature`,i) for feature in features])
        )
 ❸ for i in xrange(0,NUM_MINHASHES,SKETCH_RATIO):
      ❹ sketch = murmur.string_hash(`minhashes[i:i+SKETCH_RATIO]`)
        sketches.append(sketch)
    return np.array(minhashes),sketches
```

Building a persistent Malware Similarity Search System

```
def store_sample(path):
    """
    Function that stores a sample and its minhashes and sketches in the
    'shelve' database
    """
  ❶ db = get_database()
  ❷ features = getstrings(path)
  ❸ minhashes,sketches = minhash(features)

  ❹ for sketch in sketches:
        sketch = str(sketch)
      ❺ if not sketch in db:
            db[sketch] = set([path])
        else:
            obj = db[sketch]
          ❻ obj.add(path)
            db[sketch] = obj
        db[path] = {'minhashes':minhashes,'comments':[]}
        db.sync()

    print "Extracted {0} features from {1} ...".format(len(features),path)
```




```
def comment_sample(path):
    """
    Function that allows a user to comment on a sample.  The comment the
    user provides shows up whenever this sample is seen in a list of similar
    samples to some new samples, allowing the user to reuse their
    knowledge about their malware database.
    """
    db = get_database()
    comment = raw_input("Enter your comment:")
    if not path in db:
        store_sample(path)
    comments = db[path]['comments']
    db[path]['comments'] = comments
    db.sync()
    print "Stored comment:", comment
```

Building a persistent Malware Similarity Search System

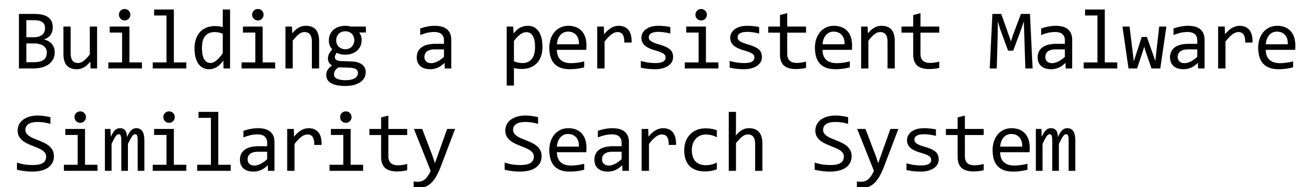
```
def search_sample(path):
    """
    Function searches for samples similar to the sample provided by the
    'path' argument, listing their comments, filenames, and similarity values
    """
    db = get_database()
    features = getstrings(path)
    minhashes, sketches = minhash(features)
    neighbors = []

❸   for sketch in sketches:
        sketch = str(sketch)

        if not sketch in db:
            continue

      ❹ for neighbor_path in db[sketch]:
            neighbor_minhashes = db[neighbor_path]['minhashes']
            similarity = (neighbor_minhashes == minhashes).sum()
            / float(NUM_MINHASHES)
            neighbors.append((neighbor_path, similarity))

    neighbors = list(set(neighbors))
❺   neighbors.sort(key=lambda entry:entry[1], reverse=True)
    print ""
    print "Sample name".ljust(64), "Shared code estimate"
    for neighbor, similarity in neighbors:
        short_neighbor = neighbor.split("/")[-1]
        comments = db[neighbor]['comments']
        print str("[*] "+short_neighbor).ljust(64), similarity
        for comment in comments:
            print "\t[comment]",comment
```



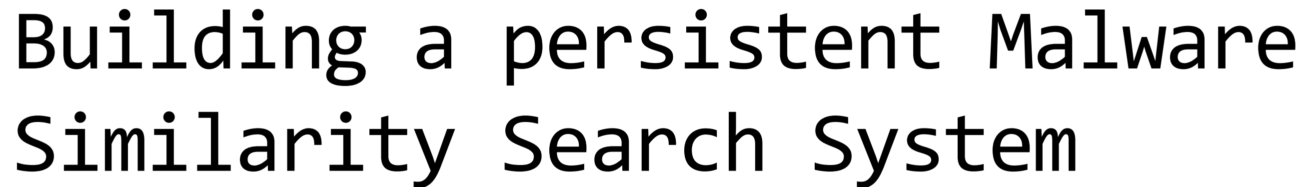

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="""
Simple code-sharing search system which allows you to build up
a database of malware samples (indexed by file paths) and
then search for similar samples given some new sample
"""
    )

    parser.add_argument(
        "-l", "--load", dest="load", default=None,
        help="Path to malware directory or file to store in database"
    )

    parser.add_argument(
        "-s", "--search", dest="search", default=None,
        help="Individual malware file to perform similarity search on"
    )

    parser.add_argument(
        "-c", "--comment", dest="comment", default=None,
        help="Comment on a malware sample path"
    )

    parser.add_argument(
        "-w", "--wipe", action="store_true", default=False,
        help="Wipe sample database"
    )
```



Building a persistent Malware Similarity Search System

```
args = parser.parse_args()
if args.load:
    malware_paths = []  # where we'll store the malware file paths
    malware_features = dict()  # where we'll store the malware strings
    for root, dirs, paths in os.walk(args.load):
        # walk the target directory tree and store all of the file paths
        for path in paths:
            full_path = os.path.join(root,path)
            malware_paths.append(full_path)

    # filter out any paths that aren't PE files
    malware_paths = filter(pecheck, malware_paths)

    # get and store the strings for all of the malware PE files
    for path in malware_paths:
        store_sample(path)

if args.search:
    search_sample(args.search)

if args.comment:
    comment_sample(args.comment)

if args.wipe:
    wipe_database()
```