Appendix: Some real-world and research uses of hashing

CS138 Winter 2017

Prof. Mike Godfrey
University of Waterloo

(Review) Hash tables & functions

- A hash function maps a key type into a range of positive integers 0..K-1
  - The key type could be just about any type: int, string, pointer type, struct type
  - Hash functions must be deterministic, based only on properties of the key values
  - A good hash function will have a fairly uniform "spread" of values over the target range 0..K-1 regardless of lumpiness in the input key values
  - Hash functions do not have "locality":
    • If key1 is close to key2, then hash(key1) will probably not be close to hash(key2)
    • That's a good thing, BTW

(Review) Important characteristics of a hash function

1. Deterministic, based on key value [absolute requirement!]
   - Must get the same answer for the same input!
   • Else lookup later will fail
   - So we can't just assign a random number tag to each incoming element and spray data at the buckets based on that; instead we want to take some intrinsic property of the input data that feels random and use that to select the bucket
2. Good, even-ish spread of results over buckets ("uniformity")
3. Cheap to compute
4. Supports a variable range
   i.e., easy to adapt if # of buckets changes
(Review) Hash tables & functions

• A hash table consists of a vector of K buckets, a hash function, and a collision resolution strategy
  – We have seen two strategies:
    • Closed hashing with linear probing
    • Open hashing with chaining
  – A good hash table implementation is \( O(1) \) for lookup/insert/delete

• Usually, a hash table provides a mechanism to iterate through all of the elements, but there is no assumption about how this order of iteration relates to the key order
  – If you need a sorted container, use a self-balancing BST (e.g. a red-black tree) instead

(Review) Other uses of hashing

• Error correction:
  – It’s slow to send file over a noisy wire; so take a hash of the file at either end
  – If the hashes match then with very high probability, the send worked perfectly

• Plagiarism detection:
  – First, normalize the source code to remove details that might differ that you don’t care about (e.g., remove comments, boiler-plating, reformat to a common whitespace convention)
  – Then, creates hashes at some granularity (e.g., one per LOC), and compare sets of hashes. If two files have a lot of hashes in common, do a more expensive comparison (e.g., manual) on the candidates
  – This is much cheaper than comparing program graphs

(Review) Other uses of hashing

• Basically, a hash function can reduce a large digital "entity" to a relatively small number that is unique with high probability
  – Effectively, you are reducing the dimensionality of the data space ☺
  – So if you want to compare a large set of large entities against each other for uniqueness, then it’s much simpler to take the hash of each and just compare the hashes

Other uses of hashing

• Spell checking
  – Hash a dictionary! Instant spell checker!

• Password validation
  – In the old days of Unix, the hash of your password was stored in a globally readable plain text file /etc/passwd
  – You would type in your password on login, the system would hash it and see if the hash matched what was stored in /etc/passwd
  – This means it never has to store your actual password anywhere!
  – In principle, you can’t reverse engineer the original password from the hash because the hash function is "lossy"; this requires a cryptographic hash function
Other uses of hashing

• A cryptographic hash function has these properties [Wikipedia]
  – it is easy (cheap) to compute the hash value for any given message
  – it is infeasible to generate a message that has a given hash
  – it is infeasible to modify a message without changing the hash
  – it is infeasible to find two different messages with the same hash

• MD5, SHA0, and SHA1 are examples of cryptographic hash functions that are considered to be "broken" or "probably breakable", tho they still work quite well as plain old hash functions
  – There are many other cryptographic hash functions (e.g., SHA2) that appear to be OK for now
  – We have an awesome research group at UW-CS in this area: CRYSP (CRYptography, Security, and Privacy)
  – Important news about SHA-1 (2017):
    https://phys.org/news/2017-02-cwi-google-collision-industry-standard.html

A research application of hash functions

• Relax!
  – This will give you a taste of what software engineering research is like

• If you're interested, the paper can be found here:
“Provenance”

A set of documentary evidence pertaining to the origin, history, or ownership of an artifact.

[From “provenir”, French for “to come from”]

• Originally, used for art / antiques, but now used in science and IT:
  – Data provenance / audit trails
  – Component provenance for security
  – ... but what about source code artifacts?

Who are you?

Alphonse Bertillon (1853-1914)

"[Holmes'] conversation, I remember, was about the Bertillon system of measurements, and he expressed his enthusiastic admiration of the French savant."

[A. Conan Doyle, The Memoirs of Sherlock Holmes]

Bertillonage metrics

1. Height
2. Stretch: Length of body from left shoulder to right middle finger when arm is raised
3. Bust: Length of torso from head to seat, taken when seated
4. Length of head: Crown to forehead
5. Width of head: Temple to temple
6. Length of right ear
7. Length of left foot
8. Length of left middle finger
9. Length of left cubit: Elbow to tip of middle finger
10. Width of cheeks
Who are you?

Forensic Bertillonage

• Some problems ...
  – Equipment was cumbersome, expensive, required training
  – Measurement error, consistency
  – The metrics were not independent!
  – Adoption (and later abandonment)

• ... but overall it was a big success!
  – Quick and dirty, and a huge leap forward
  – Some training and tools required but could be performed with technology of late 1800s
  – If done accurately, could quickly narrow down a very large pool of mugshots to only a handful

"Software Bertillonage"

• We want quick & dirty ways investigating the provenance of a function (file, library, binary, etc.)

  – Who are you, really?
    • Entity and relationship analysis

  – Where did you come from?
    • Evolutionary history

  – Does your mother know you’re here?
    • Licensing

A story about using hash functions

• Consider this problem:
  – Java applications often ship with embedded third-party libraries to avoid DLL-hell
    • But sometimes these libraries contain security holes or are otherwise outdated

• So we would like to be able to know which versions of which libraries a given application contains
  – The e-commerce industry has an explicit requirement that all version of all libraries must be documented
  – Unfortunately, this info is often stripped out or wrong or was never there
What to do?

• Suppose we have an e-commerce application that contains a bunch of libraries whose version identifiers we are unsure of

• Suppose also that we had access to a master repository of all versions of all commonly used Java libraries

• What would be an efficient way of finding the version numbers?
  – “Boil the ocean” and hash each library source version in the master repos; this only needs to be done once per version
  – Then hash the libraries included in the app, and compare those hash values to the repos
  – This would work, assuming we have a full record of all versions with full source code

Maven2: The master repository

• Maven2 ([http://maven.apache.org](http://maven.apache.org)) is a huge repository of Java jar files
  – A jar file is a container for one or more Java libraries
    • Could contain source code, or byte code, or both

• Maven2 is unorganized!
  – Anyone can commit what they think might be useful
  – There is a lot of duplication, and more binaries than source

• At time work was done (2011), Maven 2 looked like this:
  – 150 GB of jars, zips, tarballs, etc.,
  – 130,000 binary jars (75,000 unique)
  – 26M .class files, 4M .java source files (incl. duplicates)
  – Archives contain archives: 75,000 classes are nested 4 levels deep!

Problem #1

• Incomplete version histories. Would like to find “near misses”, not just exact matches.

• Solution:
  – Instead of hashing whole files, hash just the method signatures (using the SHA1 hash function), then compare sets of hashes

\[
sim(A,B) = \frac{|\theta(A) \cap \theta(B)|}{|\theta(A) \cup \theta(B)|}
\]
Problem #2

- Sometimes we have a binary (byte code) but no source code for a library version

- Solution:
  - We could take hash of binary files, but different compilers may produce different binaries so this is not reliable
  - Instead, we use a decompiler to generate source, then normalize it
  - Then we apply previous technique ... but only to the method signatures

Comparing hashes (and sets of hashes) of method signatures is fast and simple
  - Need to “boil the ocean” only once to create master repository of hashes
  - Generating hashes of the candidate app is fast
  - Effectiveness of the search depends on completeness of the repository

Testing the extractor, sampling the data

- We randomly picked 1000 binary jars for which there was also a source jar in Maven2
  - # of classes per binary archive: median: 5, max: 2138

- Binary-to-binary matching (bin2bin):
  - Each binary archive matched itself
  - # of exact matches in Maven (due to duplication or unchanging API)
    - median: 5, max: 487

- Binary-to-source matching (bin2src):
  - Correct match was among top matches (median:4, max: 158): 966 times
  - Something else was a better match (test classes): 30 times
  - No matches suggested (compiler/extractor issues): 4 times

Empirical study

**RQ1:** How useful is the signature similarity index in finding the original *binary* archive for a given binary archive?
- Found exact match or correct product 81 / 84 times (96.4%)

**RQ2:** How useful is the signature similarity index at finding the original *sources* for a given binary archive?
- Found exact match or correct product 57 / 84 times (67.9%)
Further uses

1. Used the version info from extracted from e-commerce app (84 jar files) to perform audits for licensing and security
   - One jar changed open source licenses (GNU Affero, LGPL)
   - One jar version was found to have known security bugs

2. When did Google Android developers copy-paste httpclient.jar classes into android.jar?
   - And how much work would it be to include a newer version?
   - We narrowed it down to two likely candidates, one of which turned out to be correct.

Summary

• Both of these techniques involve taking a quick hash or "snapshot" of the candidate
  - Bertillonage: Look at all the mug shots that map to the same bucket of body measurements
  - "Software Bertillonage": Look at the repository members that contain many of the same hashes of method signatures

• In both cases, we use a cheap hash-like technique to narrow the candidate set down from "a gazillion" to "a few", which can then be compared using a more expensive approach such as manual examination

Appendix: Some real-world and research uses of hashing

You don't need to study these slides for the mid-term/exam

And in conclusion ...

• Don't be a stranger!
  - If you have feedback — even years later — I would be delighted to hear it!

• Take some risks!
  - You typically learn a lot more from failure than success
  - You have time to fail several times before you are tied down to a family, mortgage, etc.
  - Do something interesting and worthwhile
  - Pay back into the community
  - Most people don't like their jobs. Create a job you want, and it won't feel like working.
And in conclusion ...

• Have a good life 😊

Fin