sdhash tutorial

Release 0.8

Vassil Roussev & Candice Quates

August 06, 2013

CONTENTS

1 Introduction **3**
- 1.1 Use cases . . . 3
- 1.2 Lay of the tutorial . . . 3
- 1.3 License . . . 4

2 Installation **5**
- 2.1 Download . . . 5
- 2.2 Linux . . . 5
- 2.3 Windows . . . 5
- 2.4 MacOS . . . 5

3 sdhash license **7**

4 Quick start **9**
- 4.1 Digest generation . . . 9
- 4.2 Digest comparison . . . 10
- 4.3 Threshold (`-t`) . . . 12
- 4.4 Result interpretation . . . 12

5 Understanding your options **15**
- 5.1 Block-aligned hashes (`-b`) . . . 15
- 5.2 Segmentation (`-z`) . . . 15
- 5.3 Parallel execution (`-p`) . . . 16
- 5.4 File list (`-f`) and recursion (`-r`) . . . 16
- 5.5 Generate & compare (`-g`) . . . 17
- 5.6 Standard input (`-`) and naming (`[--hash-name]`) . . . 17
- 5.7 Custom configuration (`[--config-file]`) . . . 17
- 5.8 The simple options . . . 18

6 Advanced processing **19**
- 6.1 GPU acceleration . . . 19
- 6.2 Sampling (`-s`) . . . 19
- 6.3 Indexing `[--index] [--index-dir arg]` . . . 20
- 6.4 Client/server processing . . . 21

7 Case studies **23**
- 7.1 Files vs. dumps . . . 23
- 7.2 Files vs. files . . . 23

authors Vassil Roussev, Candice Quates

date August 06, 2013

version 0.8

Contents:

CHAPTER

ONE

INTRODUCTION

`sdhash` is tool that allows two arbitrary blobs of data to be compared for similarity based on common strings of binary data. It is designed to provide quick results during the triage and initial investigation phases. It has been in active development since 2010 with the explicit goal of becoming fast, scalable, and reliable.

1.1 Use cases

There two general classes of problems where `sdhash` can provide significant benefits–**fragment identification** and **version correlation**.

In **fragment identification**, we search for a smaller piece of data inside a bigger piece of data ("needle-in-a-haystack"). For example:

- *Block vs. file correlation*: given a chunk of data (disk block/network packet/RAM page/etc), we can search a reference collection of files to identify whether the chunk came from any of them.
- *File vs. RAM/disk image*: given a file and a target image, we can efficiently determine if any pieces of the file can be found on the image (that includes deallocated storage).

In **version correlation**, we are interested in correlating data objects (files) that are comparable in size and, thus, similar ones can be viewed as versions. These are two basic scenarios in which this is useful–identifying related documents and identifying code versions.

Note: In all cases, the use of the tool is the same, only the interpretation may differ based on the circumstances.

1.2 Lay of the tutorial

This tutorial provides examples and case studies that illustrate all of the above scenarios along with guidance on how to obtain and interpret the results. We recommend that you:

0. Skim through the case studies to get ideas on where `sdhash` may fit into your workflow (15-30min);
1. Read the installation guide, and install the software (5-15min);
2. Work through the quick start guide (30-60min);
3. Quickly skim through the options–just to get an idea of what is possible (10min).

At this point, you should be ready to start trying things on your own. As you get to understand the capabilities of `sdhash` and start running on larger data sets, you will probably be motivated to come back and learn in more detail about options and advanced techniques.

We hope you find this tutorial helpful; send comments to `sdhash@roussev.net`.

–VR

1.3 License

CHAPTER

TWO

INSTALLATION

Ubuntu Linux (64-bit) is the primary development platform for `sdhash`; this means that any *alpha* stage releases may only be available on Linux. However, all beta and regular releases are supported on all three platforms.

The code base has largely been decoupled from the underlying platform dependencies by using appropriate high-level libraries. (All libraries have compatible licences–Apache, BSD, MIT, or similar–no GPL.) One of the big benefits is that the code has become cleaner and there are no conditional compilation statements.

2.1 Download

Everything related to `sdhash`–code, documentation, presentations, etc.–is maintained at sdhash.org. The code repo is publicly available at sdhash.cs.uno.edu.

2.2 Linux

We provide 32-/64-bit both *deb* and *rpm* packages, which are tested on recent versions of Ubuntu and Fedora (as indicated). There is also a *deb* package for *Apache Thrift*, which can simplify the build for the *client/server* version.

You can also build from source and that tends to work reliably (let us know otherwise!).

2.3 Windows

Since version 2.2, we provide 32-/64-bit native executables for MS Windows, which should worked on current versions of Windows.

The Windows port of the code is built with MSVC and come as a self-contained executable with all libraries bundled in. It supports parallel processing and measured performance is comparable to Linux. Build instructions are provided, if you need to experiment.

2.4 MacOS

Mac users need to build from source (for now); we test the build process on Mac before release, and it works reliably.

CHAPTER

THREE

SDHASH LICENSE

CHAPTER

FOUR

QUICK START

`sdhash` is designed to work right out of the box with minimal number of command line parameters. At the same time, the tool provides considerable flexibility for advanced users that can be introduced incrementally, as needed.

Invoking the tool from the command line with *no arguments* shows a brief summary of available command line options (on Linux, the `man sdhash` commands provides more details):

```
sdhash 3.3 by Vassil Roussev, Candice Quates [sdhash.org] 07/2013

Usage: sdhash <options> <files>
Configuration:
  -r [ --deep ]                   generate SDBFs from directories and files
  -f [ --target-list ]            generate SDBFs from list(s) of filenames
  -c [ --compare ]                compare SDBFs in file, or two SDBF files
  -g [ --gen-compare ]            compare all pairs in source data
  -t [ --threshold ] arg (=1)     only show results >=threshold
  -b [ --block-size ] arg         hashes input files in nKB blocks
  -p [ --threads ] arg            restrict compute threads to N threads
  -s [ --sample-size ] arg (=0)   sample N filters for comparisons
  -z [ --segment-size ] arg       set file segment size, 128MB default
  -o [ --output ] arg             send output to files
  --separator arg (=pipe)         for comparison results: pipe csv tab
  --hash-name arg                 set name of hash on stdin
  --validate                      parse SDBF file to check if it is valid
  --index                         generate indexes while hashing
  --index-search arg              search directory of reference indexes
  --config-file arg (=sdhash.cfg) use config file
  --verbose                       warnings, debug and progress output
  --version                       show version info
  -h [ --help ]                   produce help message
```

Note: Whenever we discuss command line options, we will quote both the short and long form as in `-c [--compare]`.

4.1 Digest generation

Invoked with one, or more, file names, `sdhash` produces the similarity digests of each one files and prints them to standard output. Each digest is completely self-contained and is exactly one (potentially very, very long) line of printable ASCII characters. It consists of several header fields separated by semicolons, followed by base64-encoding of the (binary) digest data. (The details of the format are explained in [?].)

Example Generate the sdhash of file `014.html`:

```
$ sdhash 014.html
sdbf:03:8:014.html:38053:sha1:256:5:7ff:160:3:75:HZvUjQGMcYEfsk8IQgIggA ...
```

Lines from different files can be freely combined using standard text processing utilities. For all practical purposes, there are only two "user serviceable" fields–the file/object name (prefaced by its length) followed by the original object/file length.

Note: Minimum file size if 512 bytes; `sdhash` will skip over smaller files quietely. If you need warning about these, use the `[--verbose]` option.

Standard pathname patterns expansion (globbing) works as expected, courtesy of the shell:

Example Hash all html files in the current directory:

```
$ sdhash *.html
sdbf:03:8:014.html:38053:sha1:256:5:7ff:160:3:75:HZvUjQGMcYEfsk8IQgIggA ...
sdbf:03:8:056.html:6664:sha1:256:5:7ff:160:1:99:AAAYTGCQsiAOBEkANREgAIh ...
sdbf:03:8:057.html:15893:sha1:256:5:7ff:160:2:102:KSxoLJfAKAAaIQiRoAMZw ...
sdbf:03:8:058.html:734:sha1:256:5:7ff:160:1:11:AAAAAAAAAAAAAAAAAAAAAAQ ...
      ...
```

Note: There are practical limits to globbing pattern expansion in shell environments. In our experience, `bash` maxes somewhere between 40,000 and 50,00 files. To get around these limitations, use the `-f [ --hash-list ]` or `-r [--deep]`. See [?].

To save the result to a file you can either use output stream redirection:

```
$ sdhash *.html > html.sdbf
```

or the `-o [--output]` option (which could be anywhere in the command line):

```
$ sdhash -o html.sdbf *.html
$ sdhash *.html -o html.sdbf
```

The naming convention is to use the `.sdbf` extension (short for *similarity digest bloom filters*) for the similarity digests (`sdhash` does *not* depend on it).

Warning: If you are generating digests from Windows PowerShell, always use the `-o [--output]` option to save the output.
Windows PowerShell automatically (and unconditionally) converts output streams into 16-bit Unicode format. This prevents the tool from correctly reading the sdbf file later (it also doubles the size of the output).

In all other cases, including running from `cmd.exe` on Windows, it is safe to use output stream redirection (>) to store the output. In fact, this is measurably faster on large data sets. (The `-o` options was created specifically to deal with PowerShell without introducing platform dependencies in the code.)

Note: The suggested naming convention is to use the `.sdbf` extension–short for *s*imilarity *d*igest *b*loom *f*ilters–for the signature file. (The implementation does *not* depend on the file name.)

4.2 Digest comparison

The primary mode of use for `sdhash` is to first generate and store the hashes for the targets of interest, and then compare them. The comparison step is initiated by the `-c [--compare]` option, which can be invoked with one or two hash sets as arguments.

4.2.1 Single set comparison

In this case, the single file parameter indicates that `sdhash` should compare all unique hash pairs in the set (for **n** hashes, this yields $n \times (n-1)/2$ comparisons).

Example The following will compare all html files in the current directory. Using the sample data included with this tutorial, the output would look like this:

```
$ sdhash -o html.sdbf *.html
$ sdhash -c html.sdbf
195.html|206.html|001
201.html|206.html|007
428.html|608.html|058
428.html|607.html|069
061.html|062.html|026
060.html|059.html|031
199.html|198.html|066
...
```

The output consists of three columns separated by the *pipe* (vertical bar) symbol. The first two columns are names of the files being compared, whereas the third one gives the similarity score, which is a number between -1 and 100.

By default, only positive scores are shown.

4.2.2 Two-set comparison

In this case, *every* hash in the first set compared against *every* hash in the second one. For example, using the `html.sdbf` from above, we can compare the set against itself (twice!) using the two-set format:

```
$ sdhash -c html.sdbf html.sdbf | sort
014.html|014.html|100
056.html|056.html|100
057.html|057.html|100
059.html|059.html|100
059.html|060.html|031
060.html|059.html|031
060.html|060.html|100
061.html|061.html|100
061.html|062.html|026
062.html|061.html|026
...
```

The main use case for the two-set comparison (and `sdhash` in general) is the querying of a reference database with unknown data in an effort to find correlations:

```
$ sdhash -o unknown.sdbf unknown/*
sdhash -c unknown.sdbf reference.sdbf
```

In our example set, we could compare html versus text files:

```
$ sdhash *.html > html.sdbf
$ sdhash *.txt  > txt.sdbf
$ sdhash -c txt.sdbf html.sdbf
```

In this case, we find no matches, which is common when comparing files of different encodings. The test set includes a set of text files that have been derived from the html files in the set using the `html2text` utility, which strips away the markup and leaves the plain text. Intuitively, we would expect html files that have large chunks of plain text inside to come out in the following experiment:

```
$ sdhash *.html-txt > html-txt.sdbf
$ sdhash -c html.sdbf html-txt.sdbf
740.html|740.html-txt|002
448.html|448.html-txt|024
418.html|418.html-txt|010
136.html|136.html-txt|016
597.html|597.html-txt|064
434.html|434.html-txt|095
```

As it turns out, the last file–`434.html`–is almost all text.

4.3 Threshold (`-t`)

The *threshold* parameter instructs `sdhash` to display only results that are greater than or equal to the parameter; by default, it is set to `1`, so all positive results are shown. The proper setting of the threshold is inherently case-specific but in the following sections we give a good starting point that works for most cases.

Setting the threshold to zero (`-t 0`) will display the results of *all* comparisons; setting it to negative one (`-t -1`) will also show comparisons that have not been performed due to insufficient data (see below).

4.4 Result interpretation

As already mentioned, the result of the comparison is a number between 0 and 100; its interpretation depends on the scenario to which the tool is applied and the data encoding (file type) of the compared objects.

4.4.1 Scenarios

There are two basic usage scenarios:

- **Fragment identification**–we search for (traces of) something "small" in something bigger. This can include comparing the content of a disk block/network packet to a file, a file to a RAM capture, or file to a disk image.
- **Version correlation**–we look for similarity between two comparably-sized objects, usually files.

Note that the distinction between the two is entirely in the eyes of the user–the tool works **exactly** the same way in all cases. The distinction is only important in understanding what the result means.

For example, we could compare an executable to a raw RAM snapshot, in search for clues that the executable had been loaded. We may get a result as high as 100, which obviously does not mean that the two objects are identical; it merely reflects that `sdhash` is highly confident that large pieces of the file are present.

Another example case would be to search for known embedded images (e.g., a corporate logo) inside a collection of documents.

Version correlation is very helpful in finding files that have non-trivial amount of commonality. For example, executables tend to change on a function-by-function basis and we have found that we can quite reliably identify new versions of the files from hashes of previous versions. For data, the commonality may come from common boilerplate (html, pdf, etc.), or as a result of normal editing operation.

4.4.2 Significance

Despite its range (0-100), the `sdhash` comparison result **should not** be intepreted as a percentage of common content. Rather, it should be viewed as a confidence value that indicates how certain the tool is that the two data objects have non-trivial amounts of commonality.

The proper use of the `sdhash` score is to examine the results in descending order, until the false positives (as defined by the user) exceed the true positives by some margin.

The following is a basic guide to intepreting the results from `sdhash` comparisons. In [?], we will add some finer points and caveats:

- **Strong (range: 21-100)**. These are reliable results with very few false positives. When used to evaluate resemblance of two comparable in size objects (files) the number is *loosely* related to the level of commonality but this is not a guarantee. When used as part of a containment query (find a small object inside a bigger one), the number can vary widely depending on the particular position of the embedding. In other words the larger object may contain the small one 100% but the score may be as small as 25.
- **Marginal (11-20)**. The significance of resemblance comparisons in this range depends substantially on the underlying data. For many composite file types (PDF, MS Office) there tends to be some embedded commonality, which is a function of commonly used applications leaving their imprint on the file; we've observed this on occasion even with JPEG files that contain lots of (Adobe Photoshop) metadata. In that sense, the tool is not wrong but the discovered correlation is usually not of interest. Other embedded artifacts, such as fonts, are also among the discovered commonalities but are rarely significant. For simpler file types, results in this range are much more likely to be significant and should be examined in decreasing order until the false positives start to dominate.
- **Weak (1-10)**. These are generally weak results and, typically, most would be false positives. However, when applied to simple file types, such as text, scores as low as **5** *could* be significant.
- **Negative (0)**. The correlation between the targets is statistically comparable to that of two blobs of random data. Special care needs to be taken when comparing large targets to each other as discovered commonality could be avaraged out to zero. For example, if two 100GB have 1GB in common, the tool will discover that fact but when averaged with the results from the remaining 99GB, the final score will almost certainly be zero, and definitely no more than one.
- **Unknown (-1)**. This is a rare occurance for files above 4KB unless they contain large regions of low-entropy data. Recall that the absolute minimum file size that `sdhash` will consider hashing is 512 bytes. (If a case requires the comparison of lots of tiny files, `sdhash` is likely the wrong tool.)

Warning: A result of 100 does not guarantee that two objects are identical. Use a crypto hash to establish identity. (We plan to incorporate a crypto hash to test for identity, starting with version 4.0.)

CHAPTER

FIVE

UNDERSTANDING YOUR OPTIONS

Once you are comfortable with the basics, it is time to work through the complete set of features that `sdhash` offers. We will still stick to the command-line utility–client/server operation is discussed in the next chapter.

5.1 Block-aligned hashes (`-b`)

In the baseline `sdhash` algorithm, the digests are generated sequentially, with each component filter representing 9-10KB, *on average*. Such digests are very suitable for objects of up to several MB. For larger targes, such as RAM/drive images, `sdhash` automatically switches to *block* mode, which allows for better parallel processing and faster comparisons.

In block mode, the target is split into fixed-size blocks (by default, 16KiB), and each one is hashed separately. The size of the block (in KiB) can be specified with the `-b [--block-size]` option. For example, it is often useful to use a block size of 4KiB for RAM images as it matches the typical page size:

```
$ sdhash -b 4 ram-capture.dd
```

The built-in threshold for transitioning to block mode is 16MiB; to *disable block mode*, specify a block size of zero:

```
$ sdhash -b 0 ram-capture.dd
```

Note: Block mode is tuned to work in the 4-16KiB range. Since the granularity of the digests goes down, the results have a somewhat higher false positive rate than the baseline algorithm.

5.2 Segmentation (`-z`)

Targets larger than 128MiB are split into 128MiB segments; that is, sdhashing a 256MiB target would result, by default, in 2 hashes.

Example Assume that `256M.rnd` contains 256MiB of random data. Then:

```
$ sdhash 256M.rnd
sdbf-dd:03:14:256M.rnd.0000M:134217728: ...
sdbf-dd:03:14:256M.rnd.0128M:134217728: ...

$ sdhash 256M.rnd > 256M.rnd.sdbf
$ sdhash -c 256M.rnd.sdbf 256M.rnd.sdbf
256M.rnd.0000M|256M.rnd.0000M|100
256M.rnd.0128M|256M.rnd.0128M|100
```

Segmentation is controlled by the `-z [--segment-size]` parameter, where the size is expressed in MiB.

Example Create & compare the digest with **no segmentation**:

```
$ sdhash -z 0 256M.rnd > 256M.rnd.sdbf
$ sdhash -c 256M.rnd.sdbf 256M.rnd.sdbf
256M.rnd|256M.rnd|100
```

Example Create & compare the digests with **segment size of 64MiB**:

```
$ sdhash -c 256M.rnd.sdbf 256M.rnd.sdbf
256M.rnd.0000M|256M.rnd.0000M|100
256M.rnd.0064M|256M.rnd.0064M|100
256M.rnd.0128M|256M.rnd.0128M|100
256M.rnd.0192M|256M.rnd.0192M|100
```

5.3 Parallel execution (`-p`)

By default, `sdhash` will detect the number of hardware-supported concurrent threads and will try to utilize all of them. If this is not the desired behavior, the `-p [ --threads ]` option provides a means to specify the exact level of parallelism to be used.

Note: For desktop systems (up to 12-16 cores), using *all* cores is easily the best choice.

For larger system (24+ cores) we find that using up to 75% of the available cores always results in improved overall throughput; beyond that, it depends on the hardware and OS.

In our experience, Intel processors provide superior performance to comparable AMD ones; this is particularly true for comparison operations, where the measured throughput is approximately *two times* that of AMD. (The latter is attributable to the much faster performance of the `POPCNT` instruction on Intel.)

[?] provides some reference numbers for throughput.

5.4 File list (`-f`) and recursion (`-r`)

As mentioned before, globbing–a service provided by the shell–has practical limitations, and if the chosen pathname expression results in an argument list that is too long, the execution will fail.

Note: If there is any chance that the set of files you are operating on will exceed 10,000, we strongly recommend that you explicitly generate the list of targets before hashing.

`sdhash` will happily take a target list in the form of a text file with one file name per line. Once the list is ready, use the `-f [--target-list]` option to digest it.

Example Hash all html files in the current subtree (Linux):

```
$ find -name *.html > file-list
$ sdhash -f file-list
sdbf:03:13:data/014.html:38053:sha1:256:5:7ff:160:3:75:HZvUjQGMcYEfsk8IQgIggA ...
sdbf:03:13:data/056.html:6664:sha1:256:5:7ff:160:1:99:AAAYTGCQsiAOBEkANREgAIh ...
...
```

It is often useful to be able to hash an entire filesystem tree; this can be accomplished with the the `-r [--deep]`.

Example Enumerate and hash *all* files recursively:

```
$ sdhash -r .
sdbf:03:13:data/014.html:38053:sha1:256:5:7ff:160:3:75:HZvUjQGMcYEfsk8IQgIggA ...
sdbf:03:13:data/056.html:6664:sha1:256:5:7ff:160:1:99:AAAYTGCQsiAOBEkANREgAIh ...
...
```

Only proper files are hashed–directories, links, and special files are skipped over. This option *is* designed to hash large directory trees and has been tested on millions of files and partitions of up to 4TB.

5.5 Generate & compare (-g)

It is sometimes useful, especially for small trial runs, to hash and compare in one step without leaving any digest file behind. The `-g` option is a combination of digest generation and single-set comparison. In other words, the command:

```
$ sdhash -g *.html
```

is logically equivalent to:

```
$ sdhash -o temp_file.sdbf *.html
$ sdhash -c temp_file.sdbf
$ rm temp_file.sdbf
```

5.6 Standard input (-) and naming ([--hash-name])

Like many other hashing tools, *sdhash* can take input from the standard input device using the **'-'** option.

Example Concatenate and hash all html files:

```
$ cat *.html | sdhash -
sdbf-dd:03:11:stdin.0000M:5784540:sha1:256:5:7ff:192:354:16384:7b:HRnEjQCMc ...
```

By default, the resulting hash is named *stdin*, which may create problems later on. To specify a custom name, use the `[--hash-name]` option.

Example Concatenate and hash all html files; name the result `html`:

```
$ cat *.html | sdhash - --hash-name html
sdbf-dd:03:10:html.0000M:5784540:sha1:256:5:7ff:192:354:16384:7b:HRnEjQCMc ...
```

Note: For hashing from `stdin`, both block mode and segmentation are enabled and *cannot* be disabled (they can still be modified within allowable ranges).

5.7 Custom configuration ([--config-file])

If you find that the `sdhash` parameters that you use often differ from the built-in defaults, you should consider saving the combination of preferred parameters in a config(uration) file. The config file is a plain text file where each line specifies a default parameter value using the long option format. E.g.:

```
block-size=16
sample-size=4
thread-count=4
threshold=10
```

Naming the file `sdhash.cfg` and placing it in the current directory will cause `sdhash` to automatically load it and use it, without any additional options. Alternatively, use the `[--config-file]` option to explicitly point to a config file.

Note: Any option you specify on the command line will always override anything specified in a config file.

5.8 The simple options

- `--separator` sets the separator character used in the comparison results. Valid arguments are `pipe` (default), `csv`, and `tab`.
- `--validate` verifies the integrity of an `sdbf` file–it reads it in and parses the digests as usual but performs no computation on them.
- `--version` prints out the version info for the `sdhash` executable. (Please include the output with any bug reports.)
- `--verbose` provides a detailed log of the operations performed by `sdhash`; all output is sent to standard error.

CHAPTER

SIX

ADVANCED PROCESSING

6.1 GPU acceleration

We have converted the sdhash comparison algorithm to utilize GPU. We currently support CUDA devices of capability 2.0 and higher, actively developed on both Tesla and Kepler platforms.

The usage differs slightly from the standard version of sdhash, as it is best used to process large quantities of sdhash files. We have been able to search a reference set of 1.6TB using this tool, using full comparisons.

```
sdhash-gpu BETA by Candice Quates, April 2013
Usage: sdhash-gpu -d dev -r ref.sdbf -t targ.sdbf
  Each reference set is processed as a whole entity.
  Target set processing compares each object to the current reference set.
Configuration:
  -d [ --device ] arg (=0)     CUDA device to use (0,1,2 etc)
  -r [ --reference-set ] arg   File or directory of fixed-size reference set(s)
                               (data size 640MB ideal, 32MB minimum)
  -t [ --target-set ] arg      File or directory of variable-sized search
                               target set(s)
  -c [ --confidence ] arg (=1) confidence level of results
  --verbose                    debugging and progress output
  --version                    show version info
  -h [ --help ]                produce help message
```

6.2 Sampling (-s)

In some common scenarios, we can speed up comparison processing 10 to 20 times by sampling the query digests. In other words, instead if using the entire object digest, `sdhash` can grab a sample of it and search, effectively, for part of the data.

Such an approach is very effective when we are trying to establish whether, for example, a file can be found in a RAM/disk capture. In such cases, we generally expect that the file is either present, or not so sampling is the perfect approach to speed up processing.

Let us work through an example. In this case, we will construct a loose (but workable) approximation of a capture using the **t5** test data set ([?]). The set consists of 4,457 data files (html, txt, doc, xls, etc.) for a total of 1.9GB of data.

We concatenate all the files to simulate the capture; then sdhash both the individual files, and the capture. Assuming all the files are in the `./t5` directory, the following `bash` script illustrates the idea:

```
$ cat  t5/* > t5.dd
$ sdhash t5/* > t5.files.sdbf
$ sdhash -z 0 t5.dd > t5.dd.sdbf
```

(We turn off segmentation (`-z 0`) to simplify the comparison output.) Now we can time the execution of the comparison and count the number of files matched with the following command:

```
$ time sdhash -c t5.files.dd t5.dd.sdbf | wc -l
```

We find that 4,456 out of 4,457 comparisons yield a positive result. On our development 24-threaded-2.9GHz-Intel server, this completes in 119.2 seconds. Then we time the sampled executions with:

```
$ time sdhash -s <n> -c t5.files.dd t5.dd.sdbf | wc -l
```

where **<n>** is *2, 4, 8, ..., 1024*. The results:

<n>	time (s)	speedup
0	119.20	1.00
2	3.50	34.06
4	5.90	20.20
8	8.90	13.39
16	13.80	8.64
32	21.70	5.49
64	32.20	3.70
128	45.60	2.61
256	63.90	1.87
512	90.20	1.32
1024	108.60	1.10

In our experience, `-s 4` gives the best trade off between speed and accuracy. In this example, we find that using a sample of four yields the following number of weak scores:

```
score =  0:  1
score < 10:  4
score < 15:  8
score < 20: 33
```

Thus, choosing a threshold of significance of 15 would yield a true positive rate of 99.8% in exchange for 20 times speedup.

On general, even if you end up using no-sampling runs, sampling gives you the option of performing a quick preliminary scan to feel out your targets.

Note: An additional benefit of sampling is that makes the timing of the execution more predictable and easier to parallelize. If we observe the load that a long comparison (such as the one in the example) places on the hardware, we notice that in the beginning all cores are 100% utilized. After a certain point, due to uneven task distribution, some cores finish their computation while other continue on. Usually, there are a few stragglers that can considerably prolong the completion of the overall workload, although 99%+ of the computation is already done.

6.3 Indexing `[--index] [--index-dir arg]`

Indexing is an experimental feature which uses a large bloom filter to represent (at the moment) 640mb groups of files at once. Hashing a large quantity of files with `[--index]` will group them into 640mb .sdbf files with corresponding .sdbf-idx index files with no guidance necessary from the user.

The indexes can be searched by passing a directory argument to `[--index-dir]` and a list of files to be searched. The index directory should contain .sdbf and corresponding .sdbf-idx files. Only search results, not hashing output, are produced by searching indexes.

Note: The GPU's best fitting reference set size is also our default size for indexed sets. Hashing a large quantity of files with the `[--index]` option will generate a directory full of optimal sized .sdbf reference files for the GPU.

6.4 Client/server processing

To be continued...

CHAPTER

SEVEN

CASE STUDIES

7.1 Files vs. dumps

Using sdhash to search for files in memory or on disk.

In this scenario we have some memory images from the M57 scenario. The chosen images cover five consecutive days. sdhash automatically breaks these memory images into 128mb chunks, as below:

```
sdbf-dd:03:23:jo-2009-12-03.ram.0896M:132108288:sha1:256:5:7ff:192:8064:16384:38:AABAAJQAAgABCAMoISg
sdbf-dd:03:23:jo-2009-12-04.ram.0000M:134217728:sha1:256:5:7ff:192:8192:16384:65:EEACBqQQYAGQEBAoAAS
sdbf-dd:03:23:jo-2009-12-04.ram.0128M:134217728:sha1:256:5:7ff:192:8192:16384:bf:mghyC9xhJ4YA8FYlMKB
```

The target machine that these images came from was suspected to be running TrueCrypt, so we can take several versions of TrueCrypt, hash them, and compare them to the memory images in sequence, like so:

```
% sdhash -c truecrypt.sdbf jo-days.sdbf -t 40   #threshold optional
TrueCrypt-6.3a/truecrypt.sys|jo-2009-12-03.ram.0256M|054
TrueCrypt-6.3a/truecrypt.sys|jo-2009-12-04.ram.0256M|051
TrueCrypt-6.3a/TrueCrypt Format.exe|jo-2009-12-03.ram.0128M|041
TrueCrypt-6.3a/TrueCrypt.exe|jo-2009-12-03.ram.0128M|043
....
```

7.2 Files vs. files

Using our set of TrueCrypt binaries from earlier, we can compare the set of installation files, unzipped, and see which parts are common to each other between versions.

```
% sdhash -c truecrypt.sdbf -t 25
TrueCrypt-6.3a/License.txt|TrueCrypt-7.0a/License.txt|087
TrueCrypt-6.3a/truecrypt.sys|TrueCrypt-7.0a/truecrypt.sys|029
TrueCrypt-5.1a/License.txt|TrueCrypt-5.1a/TrueCrypt.exe|076
TrueCrypt-5.1a/License.txt|TrueCrypt-5.1a/TrueCrypt Format.exe|076
TrueCrypt-5.1a/License.txt|TrueCrypt Setup 5.1a.exe|055
TrueCrypt-5.1a/TrueCrypt.exe|TrueCrypt-5.1a/TrueCrypt Format.exe|037
TrueCrypt-6.3a/TrueCrypt Format.exe|TrueCrypt-6.3a/TrueCrypt.exe|036
TrueCrypt-6.3a/TrueCrypt Format.exe|TrueCrypt-7.0a/TrueCrypt Format.exe|026
TrueCrypt-7.0a/TrueCrypt.exe|TrueCrypt-7.0a/TrueCrypt Format.exe|027
TrueCrypt-7.0a/TrueCrypt Format.exe|TrueCrypt Setup 7.0a.exe|026
```