Covering the global threat landscape

OPTIMIZING SSDEEP FOR USE AT SCALE

Brian Wallace
Cylance, USA

Being able to find files that are similar to a particular file is quite useful, although it can be difficult to handle at scale. It can often require an infeasible number of comparisons, which need to take place outside of a database. In an attempt to make this task more manageable, I devised an optimization to ssDeep comparisons, which drastically decreases the time required to compare files.

ABOUT SSDEEP

ssDeep [1] is a fuzzy hashing algorithm which employs a similarity digest in order to determine whether the hashes that represent two files have similarities. For instance, if a single byte of a file is modified, the ssDeep hashes of the original file and the modified file are considered highly similar. ssDeep scores range from zero (no similarity or negligible similarity) to 100 (very similar, if not an exact match).

ssDeep works by computing a fuzzy hash of each piece of data supplied to it (string/file/etc.). Most implementations of ssDeep refer to this computing of the fuzzy hash as 'compute'. The output of this compute function is an ssDeep hash, which looks like the following:

```
768:v7XINhXznVJ8CC1rBXdo0zekXUd3CdPJxB7mNmDZkUKMKZQb
FTiKKAZTy:ShT8C+fuioHq1KEFoAU
```

Once we have computed hashes for more than one input, we can conduct the comparison method (generally referred to in implementations as 'compare') to compare the two hashes. This similarity comparison is done completely independently of the files the hashes are based on. This allows for simple high-level comparisons without the need to compare each file byte by byte.

ssDeep is useful when searching for similar files. For instance, two malware samples generated by the same builder which inserts configuration statically into a stub sample, may be easy to identify as having a high similarity.

In the past, I have used ssDeep to preprocess a large number of samples. For instance, during *Cylance*'s Operation Cleaver investigation [2], there were an almost insurmountable number of malware samples to reverse engineer. A number of methods were required to reduce the sample set into clusters. One of the methods used was ssDeep. Using ssDeep clustering, I was able to see which files were similar, making it simpler to determine which samples were from the same family, as well as identify when a sample was embedded in another sample.

There are services that utilize ssDeep. These services tend to supply limited ssDeep functionality, such as reduced searching or no automated queries. The likely reason for this is how ssDeep scales, which will be covered in the next section.

There are also alternative fuzzy hashing methods that may be worth exploring, but they are outside the scope of this article.

Scaling issues

The largest issue with ssDeep as it stands is that it does not scale particularly well. In order to compare a fixed ssDeep hash against a set of other ssDeep hashes, the ssDeep compare function must be called for each hash being tested. This means if you are comparing an ssDeep hash against 1,000 other ssDeep hashes, you need to call the ssDeep comparison function 1,000 times. This can become an issue when these hashes must be retrieved from a database, requiring all hashes to be retrieved to do a lookup of similar hashes.

Furthermore, clustering (or grouping) based on ssDeep requires every ssDeep hash to be compared against every other hash. This means that if you are clustering 1,000 ssDeep hashes, 499,500 (the number of pairs among 1,000 elements) ssDeep comparison function calls are required.

Considering these issues, it is quite clear that ssDeep becomes computationally heavy at scale. This is likely what leads to services providing limited functionality to use with ssDeep.

SCALING OPTIMIZATIONS

My methodology for optimizing ssDeep comparisons at scale focuses on reducing the number of ssDeep hashes that need to be compared, which reduces the search space. This methodology avoids the need for a custom-developed library to conduct ssDeep comparisons. It also establishes that these optimizations are for the application of ssDeep at scale, not for ssDeep itself.

In order to develop optimizations to decrease the search space for similar hashes, we need to inspect how these ssDeep hash comparisons are made. Since the source code for ssDeep is publicly available, this does not require a great deal of reverse engineering. The comparisons conducted by the fuzzy_compare function are our primary focus [3]. While I will not cover everything this function does, I will cover the relevant portions.

Testing methodology

In the following sections, I will describe the optimization methods I developed for utilizing ssDeep at scale. When generating these methods, I used an isolated testing environment that is easy to reproduce.

In order to maintain this high level of reproducibility, all benchmarks are computed in a single-threaded application being executed on an Odroid XU4, which was isolated from any networks to maintain a sanitized testing environment. No timed portion of the code relies on accessing resources on disk. In order to reproduce an environment where a database is being queried, a simple *SQLite* database, hosted completely in memory, is used. No advanced features of *SQLite* are employed, and all methods are easily available to anyone wishing to reproduce these results or optimization methods.

In order to test the optimization methods, they need to accomplish a task. For this reason, all benchmarks include two tasks that are needed to use ssDeep. One of these tasks, 'Lookup', is searching for any ssDeep hash in our database where the comparison value is greater than zero. This task is computed for 1,000 hashes in order to increase the accuracy of the benchmarks for smaller database sizes. The other task,

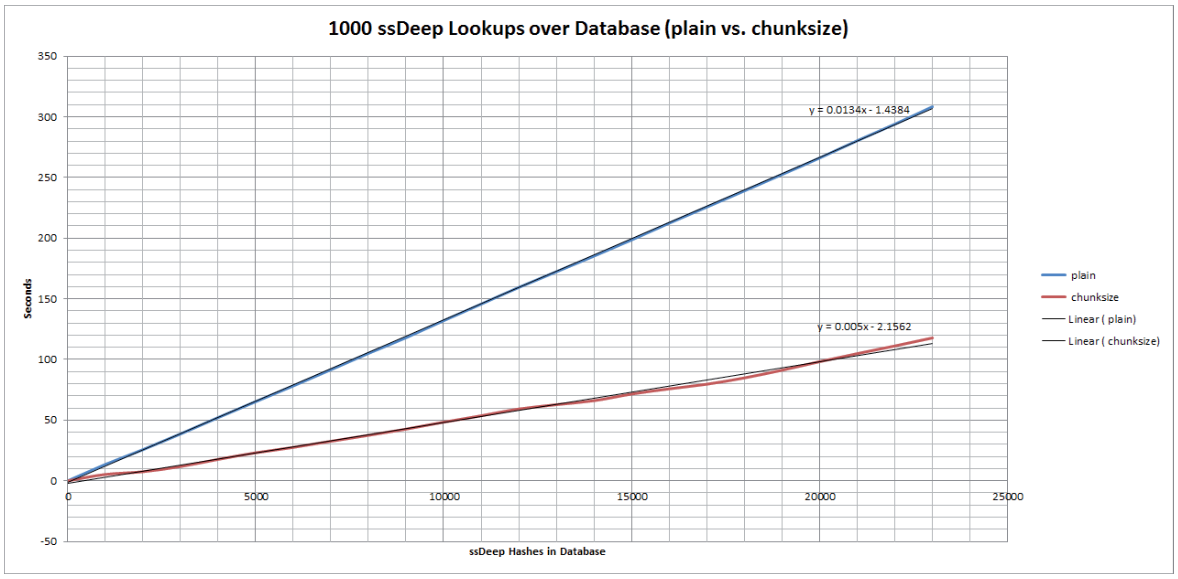

Figure 1: 1,000 ssDeep lookups over database (plain vs. chunksize).

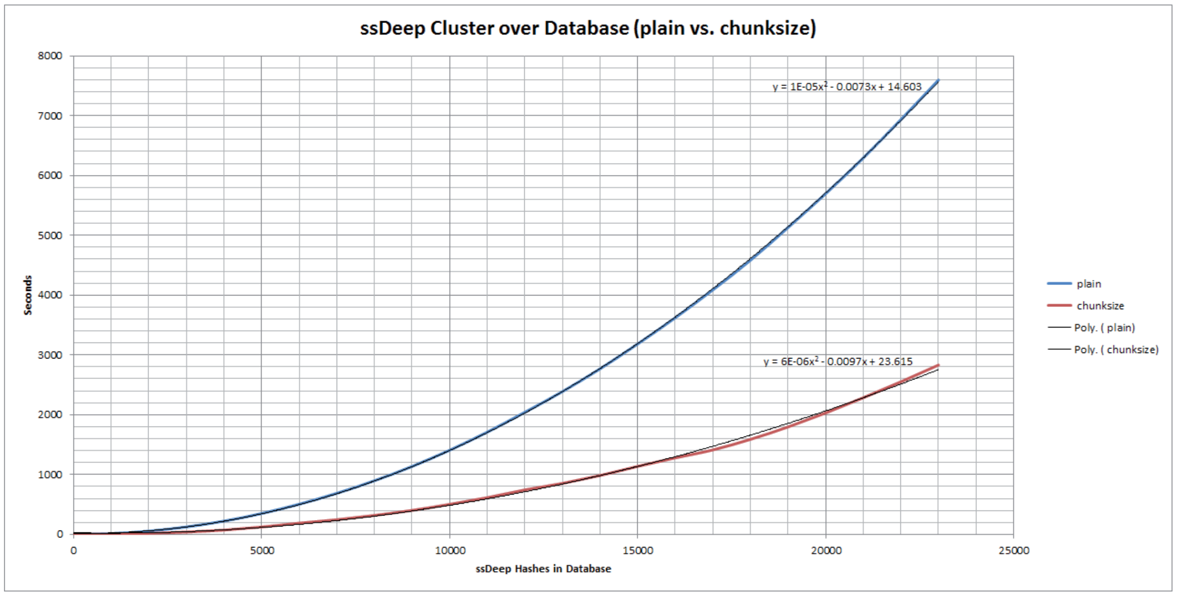

Figure 2: ssDeep cluster over database (plain vs. chunksize).

'Cluster', requires every hash in our database to be compared against every other hash. More specifically, the algorithm must return every file comparison where the value is greater than zero. The methods for clustering the resulting matrix of distance values are independent of these optimizations. Additionally, all data points are tested five times, and an average is taken over them all.

The code used to collect the benchmarks as well as the specific optimization implementations can be found at [4].

Chunksize

An ssDeep hash is formatted as follows:

```
chunksize:chunk:double_chunk
```

The chunksize is an integer that describes the size of the chunks in the following parts of the ssDeep hash. Each character of the chunk represents a part of the original file of length chunksize. The double_chunk is computed over the same data as chunk, but computed with chunksize * 2. This is done so that ssDeep hashes computed with adjacent chunk sizes can be compared. This tells us that if we are looking to perform comparisons on an ssDeep hash with chunksize=n, the comparison will not return any value other than zero unless the chunksize of the other hash is n/2, n, or 2 * n. This is our first optimization.

By only retrieving ssDeep hashes with a compatible chunksize, we reduce the number of comparisons being made unless our dataset is extremely homogenous. In order to do this, we must store our ssDeep hashes with their chunksize. For instance, we can use the following SQL schema:

```
CREATE TABLE hashes (chunksize INT, hash VARCHAR
UNIQUE);
```

With this smaller search space, we need to compute far fewer ssDeep comparisons and retrieve far fewer ssDeep hashes from our database. When benchmarks are compared with an unoptimized method, it is immediately clear that this simple optimization method is effective (see Figures 1 and 2).

Clustering optimization

We can further optimize our clustering method based on chunksize. By iterating over the chunk sizes incrementally and obtaining all ssDeep hashes of a certain chunksize, we can do our comparisons locally in an efficient manner. As long as the previous chunksize hashes are kept as well, all comparison values greater than zero can be determined.

This method works by first comparing each hash against each with the same chunksize. Then, each one is checked against each one with chunksize / 2. The previous chunksize data set retrieved is used unless a chunksize has no representative hashes. As this is done incrementally, this will compute all comparisons for chunksize / 2, chunksize and chunksize * 2 (see Figure 3).

IntegerDB

Further into the comparison process, each ssDeep hash goes through independent modifications (which will be mentioned in a later section). After these modifications, there is a check that acts as the last check before an edit distance algorithm is applied. This last check is simply checking to see if there is any seven-character string that is common between the two

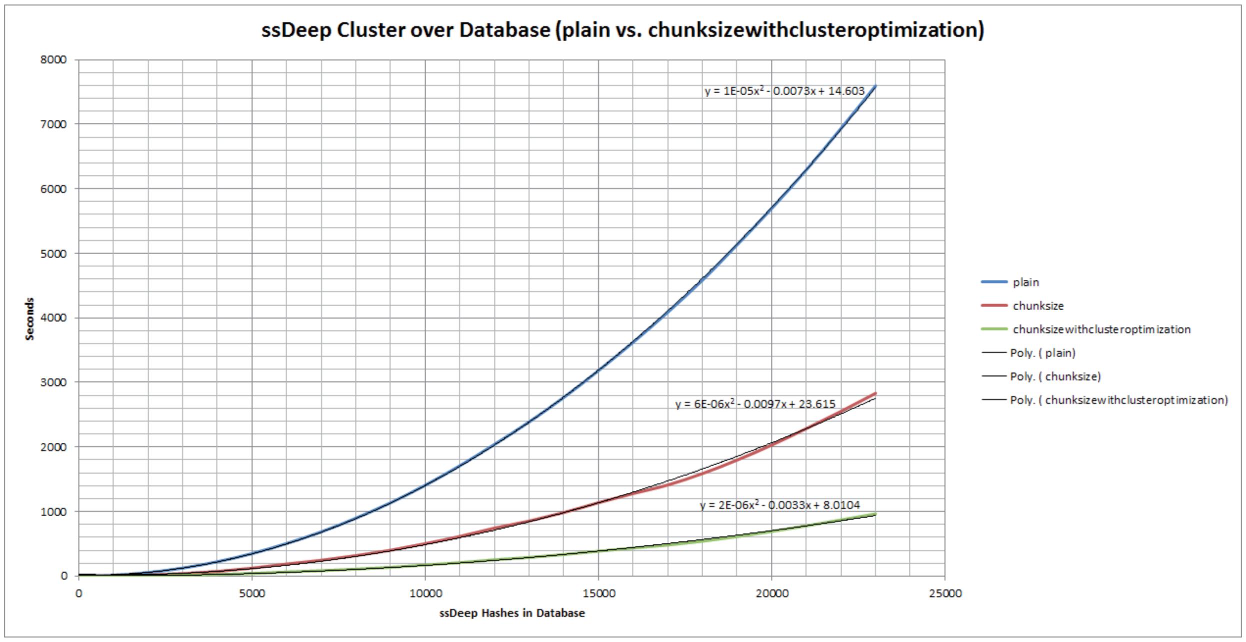

Figure 3: ssDeep cluster over database (plain vs. chunksize with cluster optimization).

hashes. If there is not, the comparison value for these two hashes will be zero. This means that if we can search over a database for any ssDeep hash with the same seven-character string, we can highly optimize our searching.

Consider the following example:

Hash 1:

```
768:v7XINhXznVJ8CC1rBXdo0zekXUd3CdPJxB7mNmDZkUKMKZQbFTi
KKAZTy:ShT8C+fuioHq1KEFoAU
```

Hash 2:

```
768:C7XINhXznVJ8CC1rBXdo0zekXUd3CdPJxB7mNmDZkUKMKZQbFTi
KKAZTV6:ThT8C+fuioHq1KEFoAj6
```

Hash 3:

```
768:t2m3D9SlK1TVYatO/tkqzWQDG/ssC7XkZDzYYFTdqiP1msdT1O
hN7UmSaED7Etnc:w7atyfzWgGEXszYYF4iosdTE1zz2+Ze
```

Since they all have the same chunksize, these three hashes can be compared. Now let's search for common seven-character strings by generating all seven-character strings for each chunk of each hash.

Hash 1:

Chunk:

```
set([['v7XINhX', '7XINhXz', 'XINhXzn', 'INhXznV',
'NhXznVJ', 'hXznVJ8', 'XznVJ8C', 'znVJ8CC', 'nVJ8CC1',
'VJ8CC1r', 'J8CC1rB', '8CC1rBX', 'CC1rBXd', 'C1rBXdo',
'1rBXdo0', 'rBXdo0z', 'BXdo0ze', 'Xdo0zek', 'do0zekX',
'o0zekXU', '0zekXUd', 'zekXUd3', 'ekXUd3C', 'kXUd3Cd',
'XUd3CdP', 'Ud3CdPJ', 'd3CdPJx', '3CdPJxB', 'CdPJxB7',
'dPJxB7m', 'PJxB7mN', 'JxB7mNm', 'xB7mNmD', 'B7mNmDZ',
'7mNmDZk', 'mNmDZkU', 'NmDZkUK', 'mDZkUKM', 'DZkUKMK',
'ZkUKMKZ', 'kUKMKZQ', 'UKMKZQb', 'KMKZQbF', 'MKZQbFT',
'KZQbFTi', 'ZQbFTiK', 'QbFTiKK', 'bFTiKKA', 'FTiKKAZ',
'TiKKAZT', 'iKKAZTy']]])
```

Double chunk:

```
set(['ShT8C+f', 'hT8C+fu', 'T8C+fui', '8C+fuio',
'C+fuioH', '+fuioHq', 'fuioHq1', 'uioHq1K', 'ioHq1KE',
'oHq1KEF', 'Hq1KEFo', 'q1KEFoA', '1KEFoAU'])
```

Hash 2:

Chunk:

```
set ['C7XINhX', '7XINhXz', 'XINhXzn', 'INhXznV',
'NhXznVJ', 'hXznVJ8', 'XznVJ8C', 'znVJ8CC', 'nVJ8CC1',
'VJ8CC1r', 'J8CC1rB', '8CC1rBX', 'CC1rBXd', 'C1rBXdo',
'1rBXdo0', 'rBXdo0z', 'BXdo0ze', 'Xdo0zek', 'do0zekX',
'o0zekXU', '0zekXUd', 'zekXUd3', 'ekXUd3C', 'kXUd3Cd',
'XUd3CdP', 'Ud3CdPJ', 'd3CdPJx', '3CdPJxB', 'CdPJxB7',
'dPJxB7m', 'PJxB7mN', 'JxB7mNm', 'xB7mNmD', 'B7mNmDZ',
'7mNmDZk', 'mNmDZkU', 'NmDZkUK', 'mDZkUKM', 'DZkUKMK',
'ZkUKMKZ', 'kUKMKZQ', 'UKMKZQb', 'KMKZQbF', 'MKZQbFT',
'KZQbFTi', 'ZQbFTiK', 'QbFTiKK', 'bFTiKKA', 'FTiKKAZ',
'TiKKAZT', 'iKKAZTV', 'KKAZTV6'])
```

Double chunk:

```
set(['ThT8C+f', 'hT8C+fu', 'T8C+fui', '8C+fuio',
'C+fuioH', '+fuioHq', 'fuioHq1', 'uioHq1K', 'ioHq1KE',
'oHq1KEF', 'Hq1KEFo', 'q1KEFoA', '1KEFoAj',
'KEFoAj6'])
```

Hash 3:

Chunk:

```
set ['t2m3D9S', '2m3D9Sl', 'm3D9SlK', '3D9SlK1',
'D9SlK1T', '9SlK1TV', 'SlK1TVY', 'lK1TVYa', 'K1TVYat',
'1TVYatO', 'TVYatO/', 'VYatO/t', 'YatO/tk', 'atO/tkq',
'tO/tkqz', 'O/tkqzW', '/tkqzWQ', 'tkqzWQD', 'kqzWQDG',
'qzWQDG/', 'zWQDG/s', 'WQDG/ss', 'QDG/ssC', 'DG/ssC7',
'G/ssC7X', '/ssC7Xk', 'ssC7XkZ', 'sC7XkZD', 'C7XkZDz',
'7XkZDzY', 'XkZDzYY', 'kZDzYYF', 'ZDzYYFT', 'DzYYFTd',
'zYYFTdq', 'YYFTdqi', 'YFTdqiP', 'FTdqiP1', 'TdqiP1m',
'dqiP1ms', 'qiP1msd', 'iP1msdT', 'P1msdT1', '1msdT1O',
'msdT1Oh', 'sdT1OhN', 'dT1OhN7', 'T1OhN7U', '1OhN7Um',
'OhN7UmS', 'hN7UmSa', 'N7UmSaE', '7UmSaED', 'UmSaED7',
'mSaED7E', 'SaED7Et', 'aED7Etn', 'ED7Etnc'])
```

Double chunk:

```
set([['w7atyfz', '7atyfzW', 'atyfzWg', 'tyfzWgG',
'yfzWgGE', 'fzWgGEX', 'zWgGEXs', 'WgGEXsz', 'gGEXszY',
'GEXszYY', 'EXszYYF', 'XszYYF4', 'szYYF4i', 'zYYF4io',
'YYF4ios', 'YF4iosd', 'F4iosdT', '4iosdTE', 'iosdTE1',
'osdTE1z', 'sdTE1zz', 'dTE1zz2', 'TE1zz2+', 'E1zz2+Z',
'1zz2+Ze'])
```

Now that we have all the seven-character strings from the chunks, we want to find any overlap between the sets with the same chunk sizes. If we have any overlap, a comparison between them will return a result greater than zero.

Hash 1 chunk & Hash 2 chunk:

```
set(['mNmDZkU', '0zekXUd', '1rBXdo0', '8CC1rBX',
'd3CdPJx', 'CC1rBXd', 'ekXUd3C', 'o0zekXU', 'PJxB7mN',
'B7mNmDZ', '3CdPJxB', 'FTiKKAZ', 'C1rBXdo', 'ZkUKMKZ',
'dPJxB7m', 'Ud3CdPJ', 'kUKMKZQ', 'XINhXzn', 'INhXznV',
'kXUd3Cd', 'znVJ8CC', 'UKMKZQb', '7XINhXz', 'nVJ8CC1',
'ZQbFTiK', 'Xdo0zek', 'JxB7mNm', 'KMKZQbF', 'XznVJ8C',
'MKZQbFT', 'QbFTiKK', 'rBXdo0z', 'CdPJxB7', 'TiKKAZT',
'NmDZkUK', 'J8CC1rB', 'VJ8CC1r', 'hXznVJ8', 'bFTiKKA',
'do0zekX', 'DZkUKMK', 'BXdo0ze', 'zekXUd3', 'mDZkUKM',
'KZQbFTi', 'XUd3CdP', '7mNmDZk', 'xB7mNmD',
'NhXznVJ'])
```

Hash 1 double_chunk & Hash 2 double_chunk:

```
set(['oHq1KEF', 'uioHq1K', 'C+fuioH', '+fuioHq',
'q1KEFoA', 'Hq1KEFo', '8C+fuio', 'T8C+fui', 'hT8C+fu',
'fuioHq1', 'ioHq1KE'])
```

Hash 1 chunk & Hash 3 chunk:

```
set([])
```

Hash 1 double_chunk & Hash 3 double_chunk:

```
set([])
```

Hash 2 chunk & Hash 3 chunk:

```
set([])
```

Hash 2 double_chunk & Hash 3 double_chunk:

```
set([])
```

With these values, we should see that the comparison between Hash 1 and Hash 2 results in a value greater than zero, but the comparisons between Hash 1 and Hash 3, and between Hash 2 and Hash 3, will result in comparisons that equal zero.

```
>>> ssdeep.compare("768:v7XINhXznVJ8CC1rBXdo0zekXUd3Cd
PJxB7mNmDZkUKMKZQbFTiKKAZTy:ShT8C+fuioHq1KEFoAU", "768:
C7XINhXznVJ8CC1rBXdo0zekXUd3CdPJxB7mNmDZkUKMKZQbFTiKKA
ZTV6:ThT8C+fuioHq1KEFoAj6")

97

>>> ssdeep.compare("768:v7XINhXznVJ8CC1rBXdo0zekXUd3Cd
PJxB7mNmDZkUKMKZQbFTiKKAZTy:ShT8C+fuioHq1KEFoAU",
"768:t2m3D9SlK1TVYatO/tkqzWQDG/
ssC7XkZDzYYFTdqiP1msdT1OhN7UmSaED7Etnc:w7atyfzWgGEXszY
YF4iosdTE1zz2+Ze")
```

```
0
>>> ssdeep.compare("768:C7XINhXznVJ8CC1rBXdo0zekXUd3Cd
PJxB7mNmDZkUKMKZQbFTiKKAZTV6:ThT8C+fuioHq1KEFoAj6",
"768:t2m3D9SlK1TVYatO/tkqzWQDG/ssC7XkZDzYYFTdqiP1msdT1
OhN7UmSaED7Etnc:w7atyfzWgGEXszYYF4iosdTE1zz2+Ze")
0
```

In order to perform this comparison optimally, we will store all seven-character string values in a database for each hash. These can be reduced to integers, as the string value consists of base64 characters, which when decoded, can optimally be represented as five-byte integers in our database. The resulting schema is as follows:

```
CREATE TABLE ssdeep_hashes (hash_id INTEGER PRIMARY
KEY, hash VARCHAR UNIQUE);

CREATE TABLE chunks (hash_id INTEGER, chunk_size
INTEGER, chunk INTEGER);
```

Now our database consists of all the integers representing seven-character strings that reside in each ssDeep hash chunk (any chunks for double_chunk have a chunksize

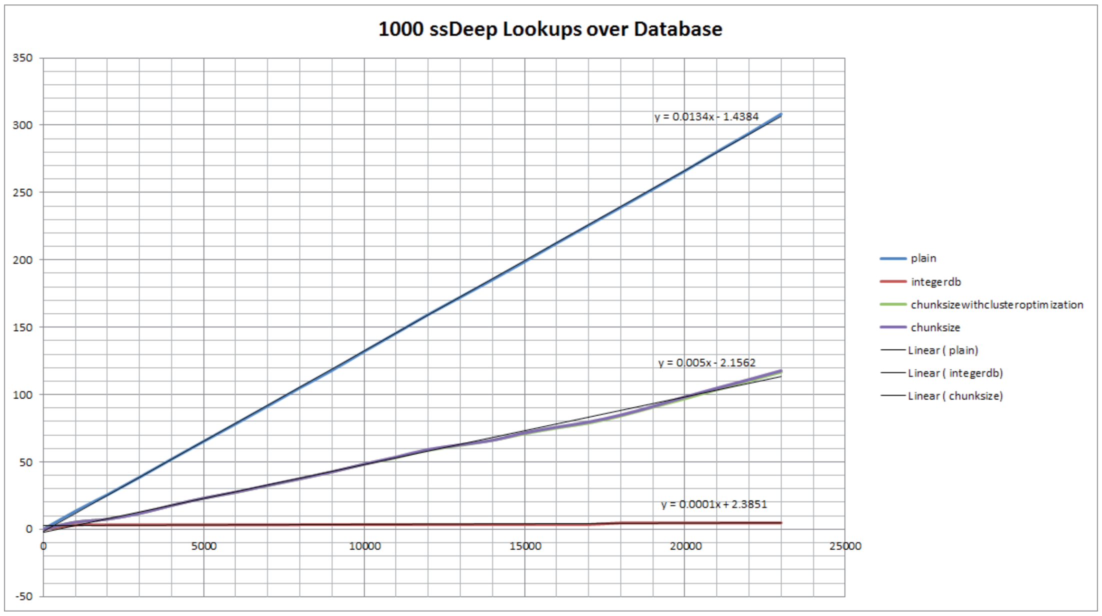

Figure 4: 1,000 ssDeep lookups over database.

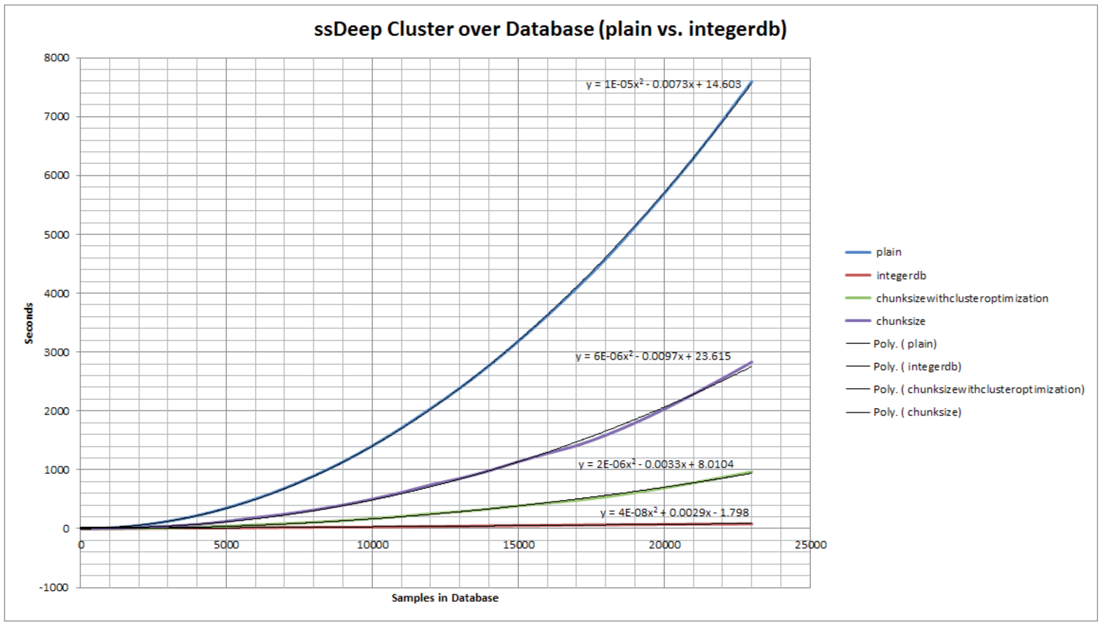

Figure 5: ssDeep cluster over database (plain vs. IntegerDB).

that represents their doubled chunk size). In order to query against this database, we need to split our query hash into integers and chunksize, then query for any hash that has the same integer and the same chunksize. Since these are queries over integers, when used with a simple index, this can be powerfully effective (see Figures 4 and 5).

This method is so effective, that in comparison to the other methods, it is difficult to visualize the curve. For this reason, additional benchmarks must be gathered on a greater scale (see Figures 6–9).

Implementation specifics

In order to assist with the implementation of the IntegerDB optimization of ssDeep comparisons, the following are implementation specific details with code examples in Python 2.7. During both the insertion of hashes into the database

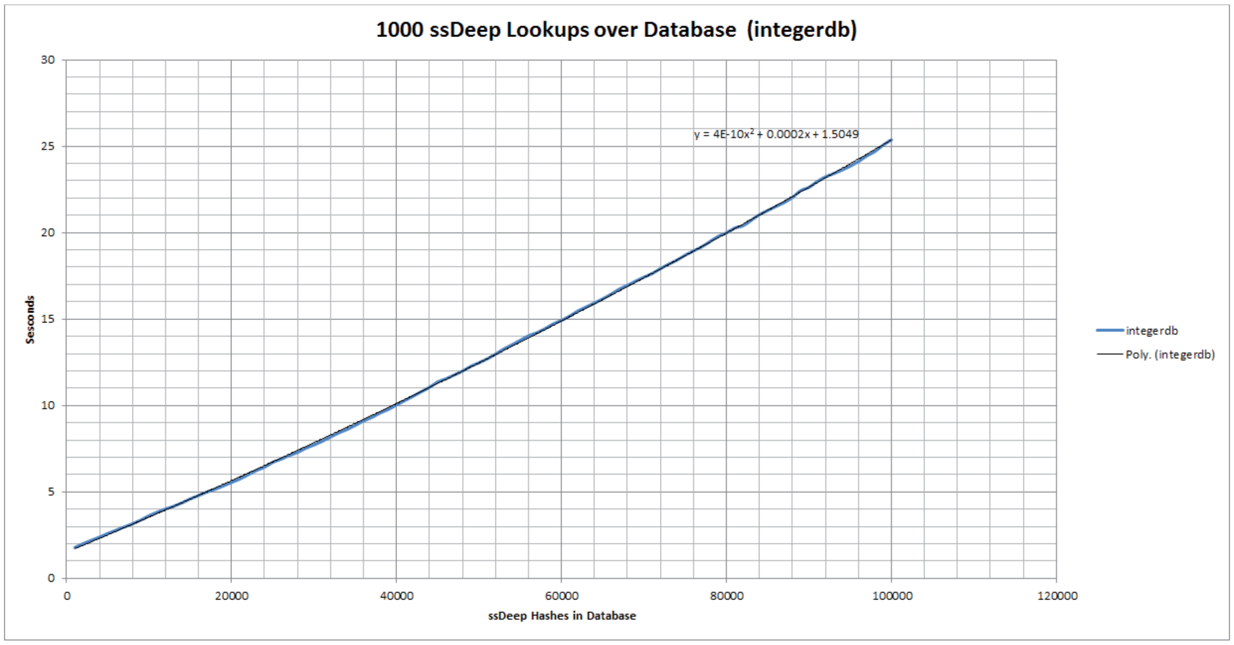

Figure 6: 1,000 ssDeep lookups over database (IntegerDB).

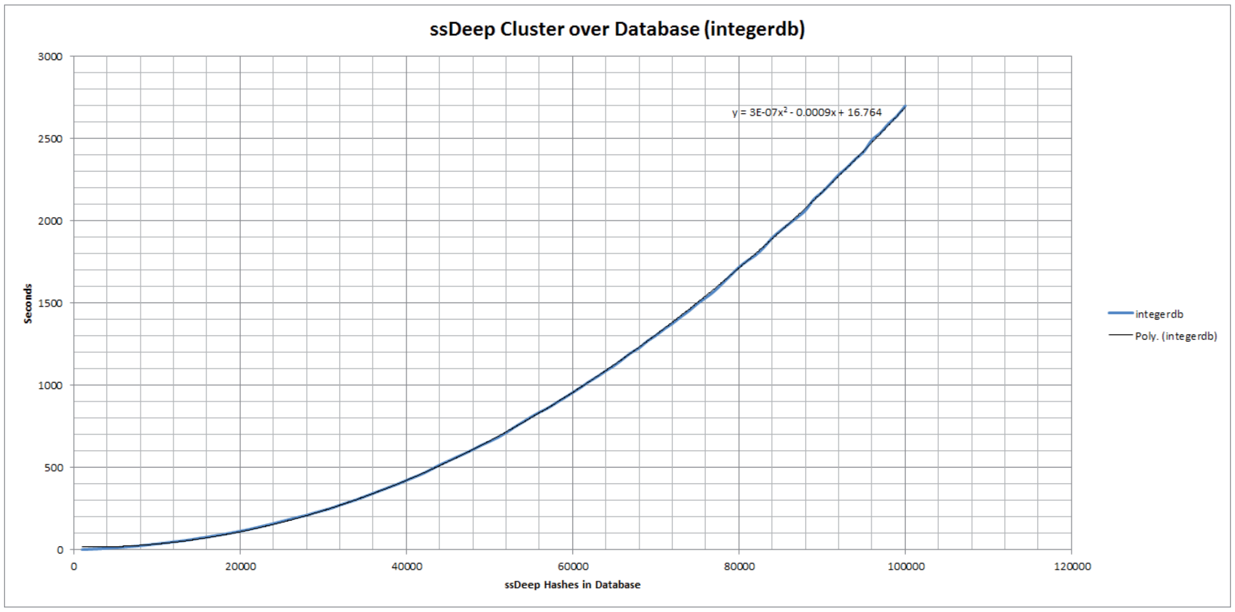

Figure 7: ssDeep cluster over database (IntegerDB).

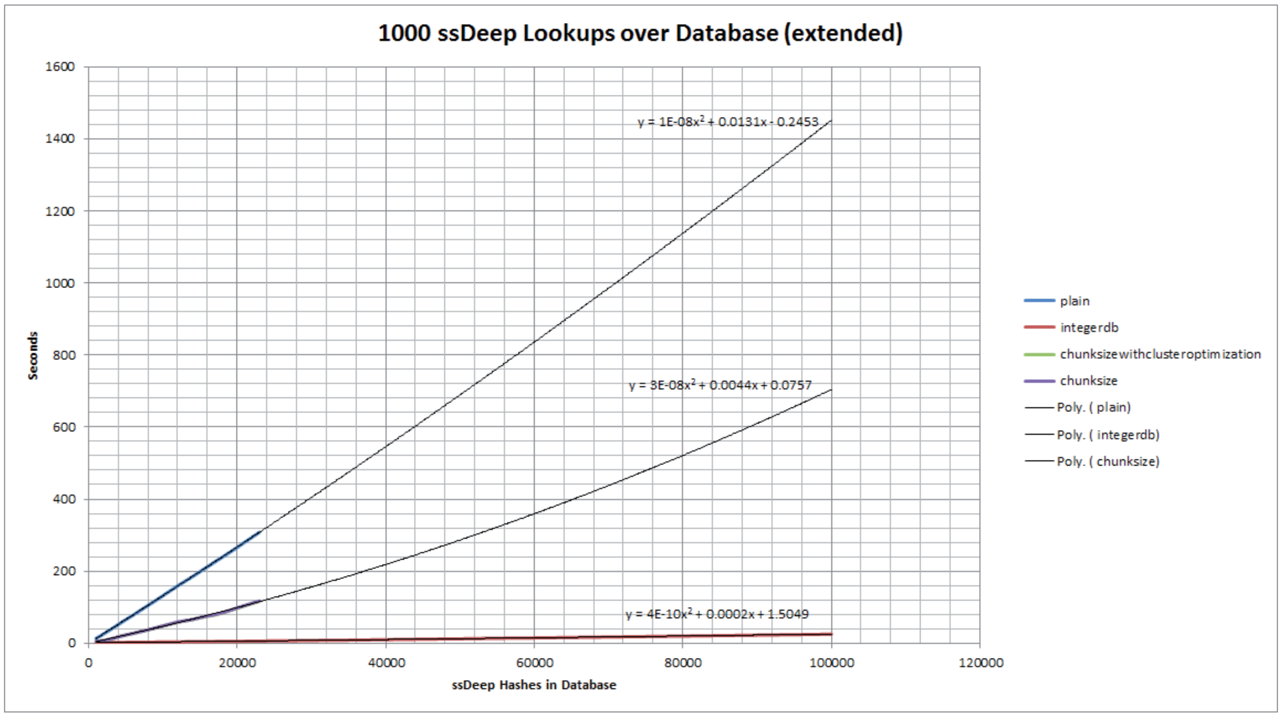

Figure 8: 1,000 ssDeep lookups over database (extended).

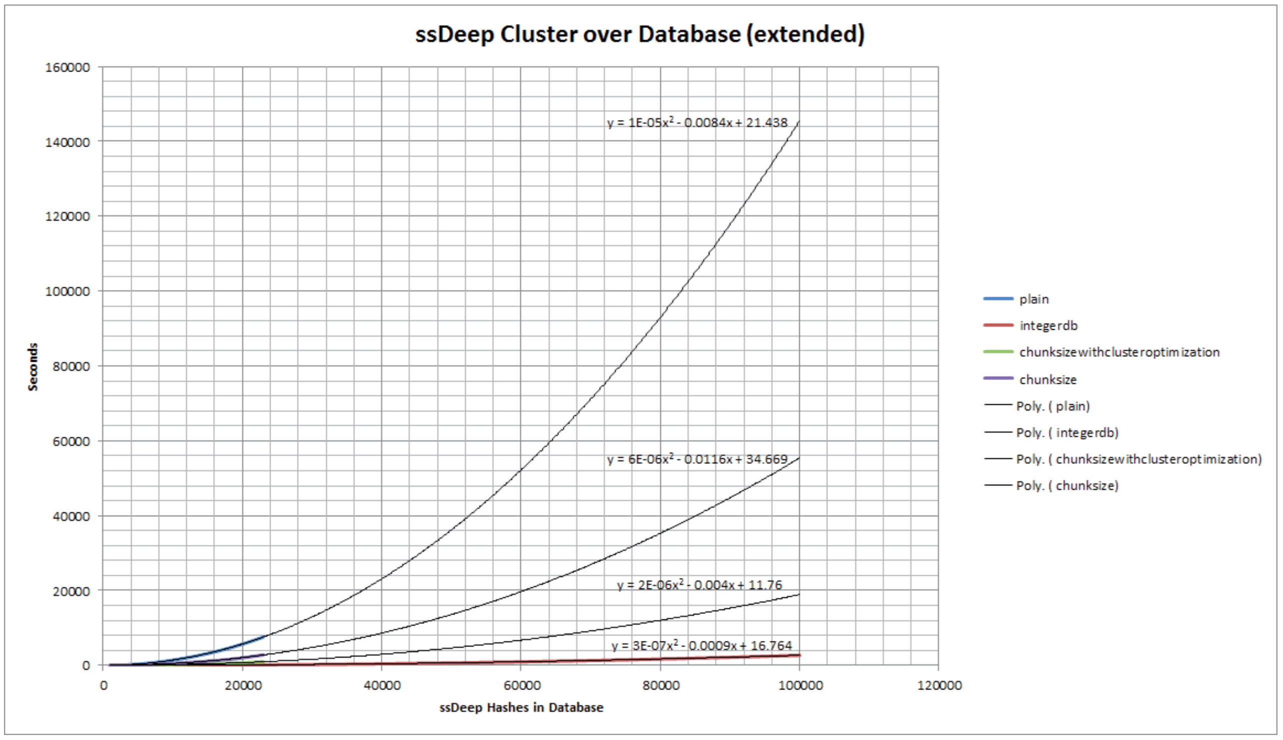

Figure 9: ssDeep cluster over database (extended).

and searching for similar results, the following preprocessing needs to be done on the ssDeep hash in question:

```
def get_all_7_char_chunks(h):
        return set((unpack("<Q", base64.b64decode(h[i:
i+7] + "=") + "\x00\x00\x00")[0] for i in
xrange(len(h) - 6)))

def preprocess_hash(h):
        block_size, h = h.split(":", 1)
        block_size = int(block_size)

        # Reduce any sequence of the same char greater
than 3 to 3
        for c in set(list(h)):
                 while c * 4 in h:
                         h = h.replace(c * 4, c * 3)

        block_data, double_block_data = h.split(":")

        return block_size, get_all_7_char_chunks(block_
data), get_all_7_char_chunks(double_block_data)
```

This code goes through a variety of steps, eventually returning the block size, all the integers from the chunk, and all the integers from the double chunk. One of the first steps in the ssDeep comparison process is to reduce any sequence of repeated characters with a length greater than three down to three. For instance, if the sequence 'aaaaa' is in either the chunk or double chunk, it will be reduced to 'aaa'. My code for this is far from optimal, but still effective.

Following this compression, we can gather all the seven-character chunks as integers. My Python code may appear a bit cryptic, so I will explain in detail. The code is essentially iterating through the entire input string, producing every seven-character string (0...6, 1...7, etc.). Once it has these strings, it then base64 decodes the string. We can do this because of the character set used in the ssDeep hash. The base64 decode decreases the size of the string to five bytes of binary data. We can convert this to a 64-bit integer, or a five-byte integer depending on language/database support, in order to speed up the rate of comparison in the database.

Finally, the resultant integers are put into a set, which effectively removes any duplicates so they do not increase the required storage space.

In order to support developers implementing this, here are test vectors to test against. The order is not important:

```
>>> preprocess_hash("384:HEOV6N0/xFXSw0x2K+PLfNDOPK2TY
WImaMsYLB3q60tL5DwpXe9hZ4ksJWoTNpyY:HEI9Xg7+P9yImaNk3q
rDwpXe9gf5xkIZ")
(
384,
set([589901095979, 642905316997, 905478568455,
235582186252, 929782237711, 340233217296,
851252513112, 221750149778, 643209346740,
128105608213, 757527782297, 953089868572,
561324516898, 680848619429, 895446934315,
326080535852, 1040326062893, 513885114415,
937588987312, 751061533361, 524104222258,
430227534644, 498584342327, 572534485560,
1085881015865, 975172384572, 98631062978,
975686749379, 329702856132, 786998125255,
755246370123, 702655530317, 43217703634,
1043551886803, 77300278103, 1044441444312,
438658545369, 166024021082, 424248880221,
819075526366, 670922140197, 745906090080,
873625707105, 568446241762, 1028060167396,
590144991069, 455890209127, 368364740072,
422543863100, 261528177643, 125293874024,
190543428819, 835680999158, 219930424056,
325920086139, 744049528956, 619266813355,
57098572287]),
set([577949007489, 150182281091, 929782237711,
822542307088, 43212569235, 733875577753, 61710615068,
526461527586, 978982065443, 1093098370981,
513885122730, 1028060167340, 945966419550,
260599074102, 702655552575, 34206553538, 331628579272,
672151957341, 643217730270, 758770030952,
591782601711, 1099381307637, 286806050806,
933755233015, 438664626168, 111251806331])
)
```

SSDC

Before discovering this optimization, I developed a tool called ssdc (ssDeep Clustering) in order to allow for quick command-line clustering of files based on their ssDeep hash. After discovering this optimization, I decided to integrate it, as it would allow for significantly larger groups of files to be clustered.

In order to maximize the benefit of IntegerDB in ssdc, no database is used, but instead, I use some native Python data structures. Additionally, all comparisons are made on insertion immediately after the hashes have been computed, allowing for a different clustering optimization from that which would be effective in a database environment.

When ssdc is executed over a set of samples, it creates a tar file that contains all the files in directories, with similar files stored in the same directory. Additionally, a GEXF file is stored in the tar file, which can be used with Gephi [5] in order to visualize the clustering.

Example

As an example of ssdc usage, I'm going to scan some sample sets of malware I have readily available:

BlackShades [6] - 134 files - Black

DiamondFox [7] - 6 files - Blue

TinyZbot [8] - 4 files - Red

After running these sample sets through ssdc, I load the resultant file_distance.gexf into Gephi, and colour all the nodes to match their malware families. I then export the graph to the image shown in Figure 10. In this image, each node (circle) represents a malware sample, and each edge (line/curve) represents an ssDeep comparison result greater than zero.

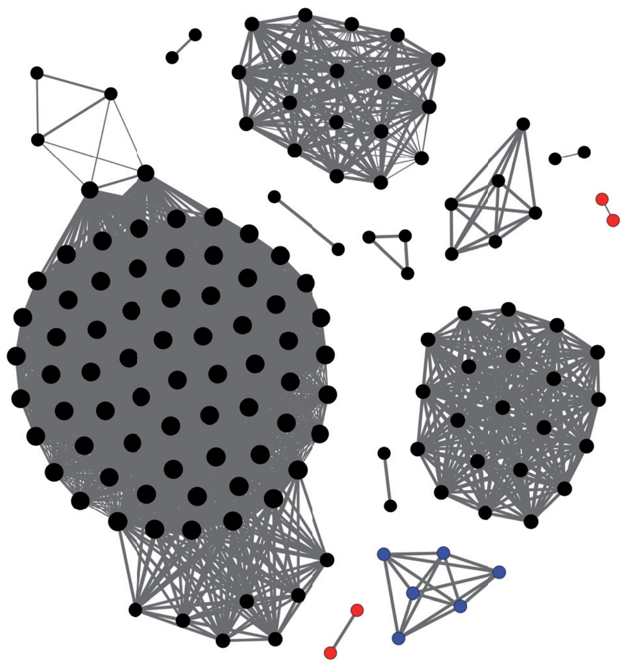

Figure 10: Each node (circle) represents a malware sample, and each edge (line/curve) represents an ssDeep comparison result greater than zero.

Consider if one did not know which families were which, how much simpler it would be to analyse a sample or two from each group, instead of having to reverse a few hundred files. You might notice that not all the samples of the same family are considered similar. Since ssDeep is performing a simple file comparison to determine if samples are similar, samples that are obfuscated may not compare with other samples of the same family that are not obfuscated, or which are obfuscated with different methods.

This tool can be downloaded from its Github repository [9].

CONCLUSION

Depending on your requirements, ssDeep can be a useful algorithm for determining the similarity of files. With naive methods, performing ssDeep comparisons does not scale, but with the IntegerDB optimization, we can utilize ssDeep on far larger scales.

REFERENCES

[1] http://ssdeep.sourceforge.net/.

[2] http://www.cylance.com/operation-cleaver/.

[3] http://sourceforge.net/p/ssdeep/code/HEAD/tree/trunk/fuzzy.c#l733.

[4] https://gist.github.com/bwall/a28733fd6890749d2413.

[5] http://gephi.github.io/.

[6] http://blog.cylance.com/a-study-in-bots-blackshades-net.

[7] http://blog.cylance.com/a-study-in-bots-diamondfox.

[8] http://www.cylance.com/operation-cleaver/.

[9] https://github.com/bwall/ssdc.

Editor: Martijn Grooten

Chief of Operations: John Hawes

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Ionuţ Răileanu

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

Consultant Technical Editor: Dr Morton Swimmer

© 2015 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com Web: https://www.virusbtn.com/