Motivation and overview

Permissive open-source licenses have been a great benefit for the IT industry: Most common problems can be addressed using high-quality libraries under permissive licenses (such as BSD or Apache) which can be easily re-used and integrated.

This has been great for everybody.

Nevertheless, statically linking third-party code requires special care: Vulnerabilities in third-party code are discovered regularly, and this implies updating your binaries. Things get harder when changes in upstream libraries need to be merged into local forks of the code. Not all organisations get this right, and even companies with mature secure development processes fumble sometimes.

For the reverse engineer and offensive researcher, identifying vulnerable statically linked software libraries is both an opportunity and a challenge:

- An **opportunity** since it can provide a way to obtain a vulnerability in a target without having to do the hard work of identifying a new one.
- A **challenge** since the available tooling for performing this task is exceptionally poor: The standard way is usually a combination of “searching for known strings”, “educated guess”, and sometimes the use of BinDiff (a tool that was designed for a very different purpose).

The technical problem can be phrased as “efficiently performing a fuzzy search into a relatively large space of possible functions”. Fuzzy search is a requirement because compiler differences, optimization changes, and code changes contribute to add “noise” to the code in question.

On the side of academic research, several interesting papers (CCS ‘16, CCS ‘17 have proposed sophisticated machine-learning-based methods to combine code embeddings with approximate nearest neighbor searches. They calculate a representation of code in R^n, and then search for nearby points to identify good candidates. While these approaches look powerful and sophisticated, public implementations do not exist, and adoption among practitioners has not happened. On the practical side, real-world use has been derived from CFG-focused algorithms such as MACHOC - but with the downside of being not tolerant to structural changes and not allowing for any “learning” of distances. Recently at (SSTIC ‘18) a neural-network based approach has been presented, with an announcement of making the code available in the next months.

This file describes FunctionSimSearch - an Apache-licensed C++ toolkit with Python bindings which provides three things:

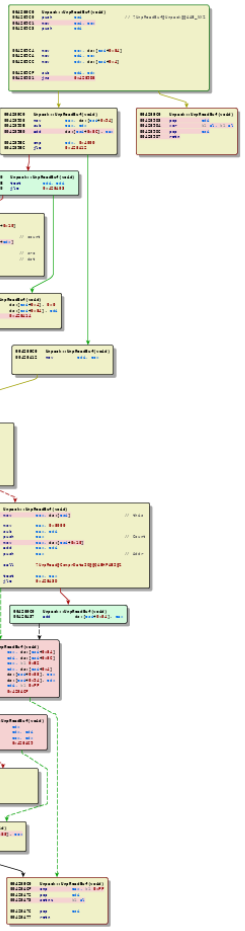
1. An efficient implementation of a hash function (based on SimHashing) which calculates a 128-bit hash from disassembled functions - and which preserves similarity (e.g. “distance” of two functions can be calculated by simply calculating the hamming distance between two hashes - which translates to two XOR and two POPCNT instructions on x64).
2. An efficient search index to allow approximate nearest neighbor search for 128-bit hashes.
3. Some supervised machine learning code that can be used to “learn” a good hash function from human-provided examples - given a set of functions that should be similar according to the hash, and a set of functions that should be dissimilar, the code learns a hash function that attempts to respect these examples.

The need for good fuzzy matching.

Every reverse engineer has encountered that different compilers and compiler settings can generate drastically different pieces of assembly. Compilers can alter the CFG significantly, move code around, and even when they do not inline aggressively or unroll loops, decisions about code duplication, instruction movement and scheduling etc. lead to very different disassemblies:

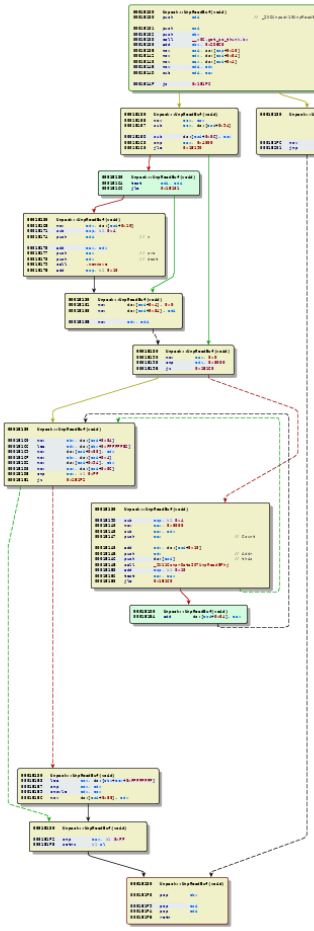
004289C0 Unpack::UnpReadBuf(void)

primary



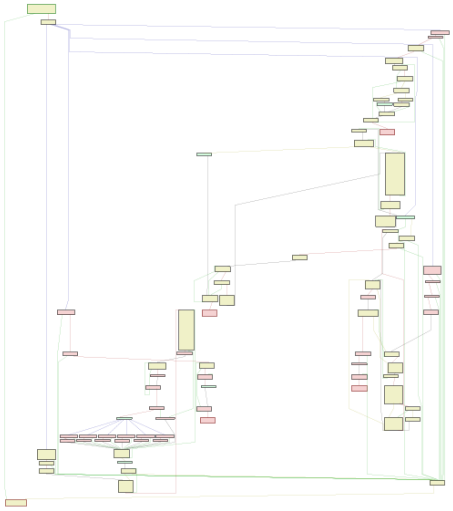
Unpack::UnpReadBuf(void) 0001E130

secondary



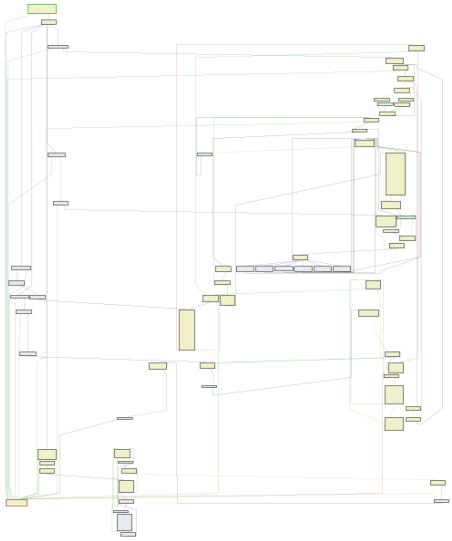
00417BB0 RarVM::ExecuteStandardFilter(VM_StandardFilters)

primary



RarVM::ExecuteStandardFilter(VM_StandardFilters) 0001152A

secondary



It is obvious that a good method to identify a function in the presence of changes is needed, and that both instruction-level and graph-level changes need to be dealt with.

Understanding the SimHash algorithm and what it provides.

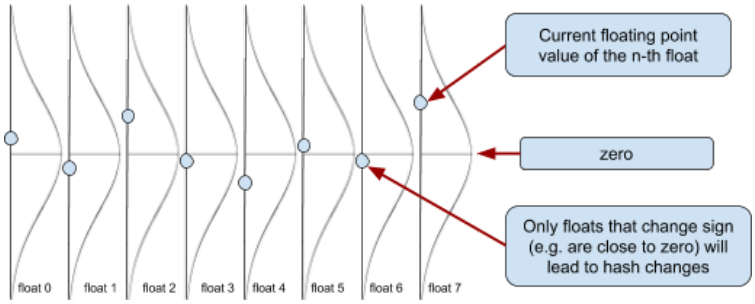
SimHashing was introduced in a paper by Moses Charikar, originally in the context of web page de-duplication. It is part of a family of algorithms and concepts called “locality-sensitive hashing”; a concept we will return to later.

The algorithm itself helps condense a set into a hash, with the property that the Hamming distance between the two hashes approximates the set similarity between the original sets.

Given an input set of features (which themselves are random 128-bit vectors, obtained through hashing), calculate a final hash value as follows:

1. Initialize a vector of 128 floating-point values to all zeroes.
2. For each feature in the input set, do: a. If bit n of the feature is 0, subtract 1 from the n-th floating-point value b. If bit n of the feature is 1, add 1 to the n-th floating point value
3. Convert the vector of floats to a 128-bit vector by mapping positive values to 1, negative values to 0.

Why does this produce a similarity-preserving hash? The intuition can be obtained by imagining what a minor change in the input set would do to the vector of floats: The values of these vectors will be approximately normally distributed with mean 0 and variance 1/4 times the number of features:



Some of the floats will be close to zero, either negative or positive. By changing the sets slightly (adding or removing a few features), there is some probability of individual floats crossing over from positive into negative territory or vice versa. This probability goes up as more of the set changes; for small changes, the odds of many bits flipping is comparatively low.

What is a good set of input features for our comparison? Ideally we would want to extract features that are representative of what we deem “similar”; e.g. two functions that are compiled from the same source code should have similar (overlapping) sets of features. It is somewhat involved to algorithmically design such features, so for the moment, the features in question are extremely simple: Subgraphs of the control-flow graph, and n-grams of mnemonics of disassembled instructions. In a naive implementation, all features have unit weight - e.g. every feature contributes the same to the final hash. This is clearly not ideal - a function prologue is not very indicative of the similarity between two functions - and we will improve this later in this post. Other non-implemented ideas for more features will be discussed at the end of the document.

A simple approximate nearest-neighbor search for hashes

With a way of calculating a similarity-preserving hash for a given input function, how do we search non-trivially sized corpora of such hashes for the "most similar" hash?

The answer lies in a second application of locality-sensitive hashing. If one can construct a family of hash functions so that the probability of two nearby points getting hashed into the same hash bucket is higher than the probability of two distant points getting hashed into the same bucket, one can construct a relatively efficient nearest-neighbor search: Simply use k different hash functions of the family to map inputs to buckets of candidates and process the candidates.

Choosing random bits as locality-sensitive hashes

Since our inputs are bit vectors, the easiest way to build such a hash function is to simply subsample bits from the vector. This has the nice property that a single random bit-level permutation of the input is enough to construct a hash family: In order to construct k different hashes, apply the bit-level permutation k times to your input and take the first few bits. Bitwise permutations on 128 bits are cheap-ish in software and close to free in hardware; the permutation chosen in the codebase should execute in ~65 cycles on a modern CPU.

Choice of data structures

The underlying data structure is an ordered collection of tuples of the form:

```
`<PermutationIndex, k-th-permutation-of-input-hash, result-id>`
```

Performing a binary search using the tuple `<k, perm_k(input) & (0xFFL << 56), 0>` will give us the hash bucket for a given permutation index and input value. We perform k such searches, and for each hash bucket we add all elements to a candidate list. The hamming distance between each candidate and the input hash is calculated, and the results can be returned in the order of their hamming distance.

A maximally memory- and cache-efficient version of this would simply use a sorted flat array / vector of such tuples; for our purposes (and for efficient insertion) the existing C++ code uses the equivalent of a std::set container, made persistent using a memory-mapped file as storage.

Learning a SimHash from examples

One of the problems with the described approach can immediately be identified: Every feature in the input set is treated with equal importance. In reality, though, features have vastly different importance. Luckily, it is easy to incorporate the importance of individual features into the calculation of a SimHash: Instead of adding +1 or -1 into the vector of floats, one could add or subtract a feature-specific weight.

But how does one infer good weights from the training data? Can we automatically "learn" what features will be preserved across compiler changes, with some predictive power?

Using the cheap gradient principle

The workhorse of modern machine learning is automatic differentiation. In simple terms, automatic differentiation provides the "cheap gradient principle" -- which can be paraphrased as “if you can calculate a function from R^n to R, you can calculate the gradient of this function with moderate overhead”. This means that if we can specify a loss function involving our weights, we can try to minimize this loss function. While we won’t have any guarantees of convergence, odds are we can learn weights from examples.

So what we need is a bunch of labelled data (ideally pairs of functions labelled "same" or "not the same"), and a good loss function.

Choosing a loss function for the SimHash distance

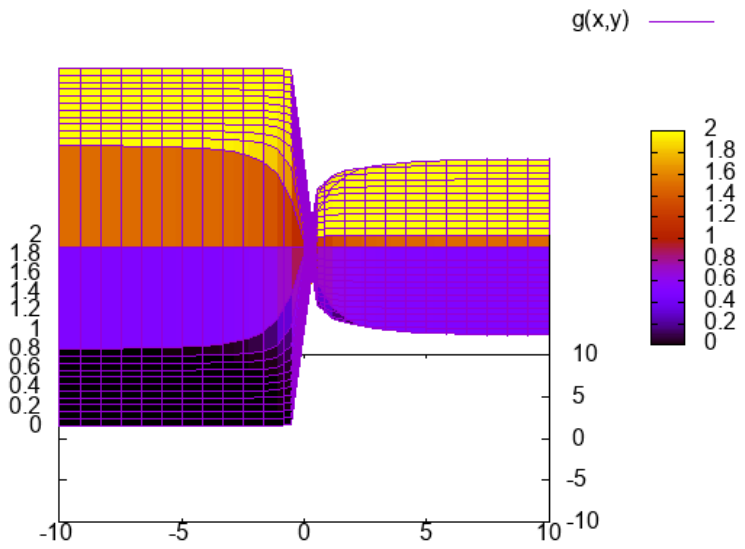
Building a loss function for our distance requires a slight bit of care. Since our final distance is a Hamming distance between two bit vectors, the gradient of this distance is likely to be zero - stepwise functions have many "flat sections" for which we cannot get a useful gradient.

The simplest idea would be to remove the last step of the hash calculation - instead of comparing the hashes that we derive from the vector-of-floats, one could measure the Euclidian distance on the final vectors-of-floats. Unfortunately, this creates "perverse incentives" for the optimizer: The simplest way to make two “similar” functions close to each other would be to shrink weights that occur in both to zero.

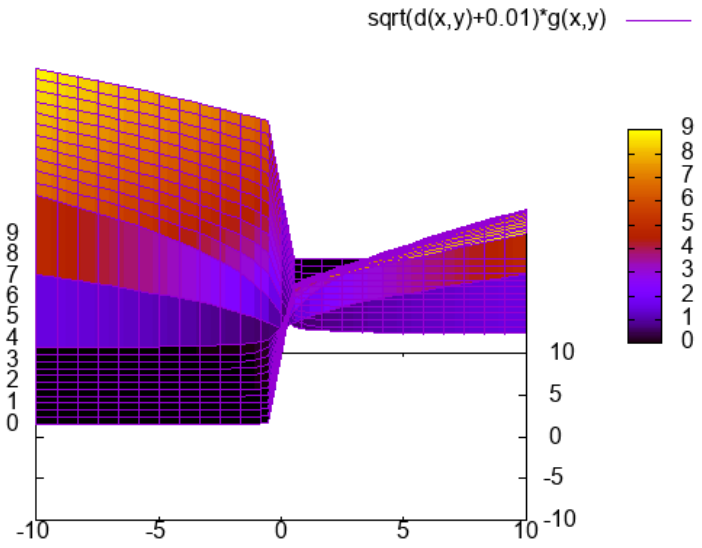
So ideally we want something that “penalizes” when pairs of similar functions with large distance and pairs of dissimilar functions with low distance.

We need a function that is positive when two real values do not have the same sign, and zero (or negative) if the two real values that have the same sign. Ideally, it should also provide a slope / incentive to move inputs in the direction of "same sign".

We start with a simple smoothed step function $$g(x, y) = \frac{-xy}{\sqrt{x^2y^2 + 1}} + 1$$



This function has high loss when the sign of x and y is different, and zero loss when it is the same. Unfortunately, it is also flat on most of the surface, and we need to somehow skew the flat regions to point into the right direction. We multiply with $d(x, y) = \sqrt{(x - y)^+0.01}$



This function satisfies our requirements: It provides a way to move parameters in the desired direction, punishes unequal signs, and has zero loss if x and y have equal sign.

In summary: For a given pair of real vectors (each obtained by calculating the hash function without the last step of converting to a binary hash) we can simply sum the loss for each vector entry. We now have a loss function that we can use to adjust our parameters from examples.

Generating training data

Generating training data should - at least in theory - be simple. It should be sufficient to compile some open-source code with a number of different compilers and compiler settings, and then parse the symbol information to create groups of "function variants" - e.g. multiple different compiler outputs for the same C/C++ function. Similarly, known-dissimilar-pairs matches can be generated by simply taking two random functions with different symbols.

Unfortunately, theory is not practice, and a number of grimy implementation issues come up, mostly around symbol parsing and CFG reconstruction.

Real-world problems: Symbols

One problem arises from the non-availability of good cross-platform tooling for parsing different versions of the PDB file format - which naturally arise when many different versions of Visual Studio are used - and the difficulty of reliably building the same open-source codebase for many different compilers. While GCC and CLANG are often drop-in-replaceable, projects that build without intervention on both Visual Studio, GCC, and CLANG are much more rare.

The (unfortunate) solution to the PDB parsing issue is “giving up” - the codebase expects the PDB information to have been dumped to a text file. More on this below.

The (unfortunate) solution to the issue of building the same codebase reliably with Visual Studio and GCC is also “giving up” - it is up to the user of FunctionSimSearch to get things built.

Other problems arise by different mangling conventions for C++ code, and different conventions in different compilers affecting how exactly a function is named. This is solved by a hackish small tool that removes type information from symbols and tries to "unify" between GCC/CLANG and Visual Studio notation. Real-world problems: Reliably generating CFGs, and polluted data sets Obtaining CFGs for functions should be simple. In practice, none of the tested disassemblers correctly disassembles switch statements across different compilers and platforms: Functions get truncated, basic blocks mis-assigned etc. The results particularly dire for GCC binaries compiled using -fPIC of -fPIE, which, due to ASLR, is the default on modern Linux systems.

The net result is polluted training data and polluted search indices, leading to false positives, false negatives, and general frustration for the practitioner. While the ideal fix would be more reliable disassembly, in practice the fix is careful investigation of extreme size discrepancies between functions that should be the same, and ensuring that training examples are compiled without PIC and PIE (-fno-pie -fno-PIE -fno-pic -fno-PIC is a useful set of build flags).

Data generation in practice

In practice, training data can be generated by doing:

```
cd ./testdata
./generate_training_data.py --work_directory=/mnt/training_data
```

The script parse all ELF and PE files it can find in the ./testdata/ELF and ./testdata/PE directories. For ELF files with DWARF debug information, it uses objdump to extract the names of the relevant functions. For PE files, I was unfortunately unable to find a good and reliable way of parsing a wide variety of PDB files from Linux. As a result, the script expects a text file with the format “<executable_filename>.debugdump” to be in the same directory as each PE executable. This text file is expected to contain the output of the DIA2Dump sample file that ships with Visual Studio.

The format of the generated data is as follows:

```
./extracted_symbols_<EXEID>.txt
./functions_<EXEID>.txt
./[training|validation]_data_[seen|unseen]/attract.txt
```

```
./[training|validation]_data_[seen|unseen]/repulse.txt
./[training|validation]_data_[seen|unseen]/functions.txt
```

Let's walk through these files to understand what we are operating on:

1. The `./extracted_symbols_<EXEID>.txt` files: Every executable is assigned an executable ID - simply the first 64 bit of it's SHA256. Each such file describes the functions in the executable for which symbols are available, in the format:

```
[exe ID] [exe path] [function address] [base64 encoded symbol] false
```

2. The `./functions_<EXEID>.txt` files: These files contain the hashes of the extracted features for each function in the executable in question. The format of these files is:

```
[exe ID]:[function address] [sequence of 128-bit hashes per feature]
```

3. The `./[training|validation]_data_[seen|unseen]/attract.txt` and `./repulse.txt` files: These files contain pairs of functions that should repulse / attract, the format is simply

```
[exe ID]:[function address] [exe ID]:[function address]
```

4. The `./[training|validation]_data_[seen|unseen]/functions.txt` files: A file in the same format as the `./functions_<EXEID>.txt` files with just the functions referenced in the corresponding `attract.txt` and `repulse.txt`.

Two ways of splitting the training / validation data

What are the mysterious `training_data_seen` and `training_data_unseen` directories? Why does the code generate multiple different training/validation splits? The reason for this is that there are two separate questions we are interested in:

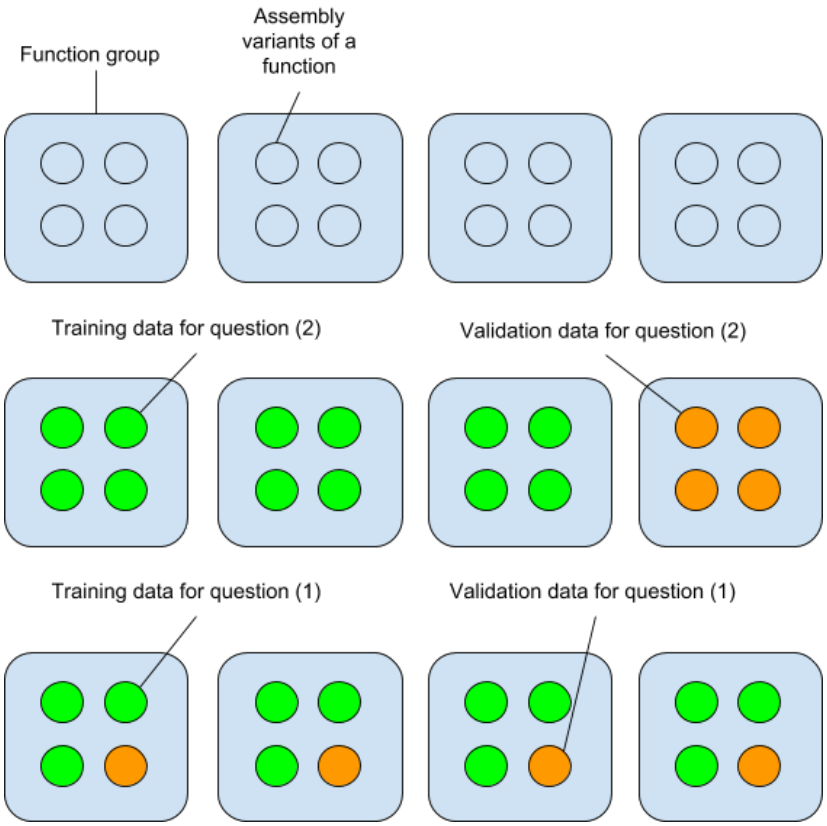
1. Does the learning process improve our ability to detect variants of a function we have trained on?
2. Does the learning process improve our ability to detect variants of a function, even if no version of that function was available at training time?

While (2) would be desirable, it is unlikely that we can achieve this goal. For our purposes (detection of statically linked vulnerable libraries), we can probably live with (1). But in order to answer these questions meaningfully, we need to split our training and validation data differently.

If we wish to check for (2), we need to split our training and validation data along "function group" lines: A "function group" being a set of variant implementations of the same function. We then need to split off a few function groups, train on the others, and use the groups we split off to validate.

On the other hand, if we wish to check for (1), we need to split away random variants of functions, train on the remainder, and then see if we got better at detecting the split-off functions.

The differences in how the training data is split is best illustrated as follows:



Implementation issues of the training

The industry-standard approach for performing machine learning are libraries such as TensorFlow or specialized languages such as Julia with AutoDiff packages. These come with many advantages -- most importantly, automated parallelization and offloading of computation to your GPU.

Unfortunately, I am very stubborn -- I wanted to work in C++, and I wanted to keep the dependencies extremely limited; I also wanted to specify my loss function directly in C++. As a result, I chose to use a C++ library called SPII which allows a developer to take an arbitrary C++ function and minimize it. While this offers a very clean and nice programming model, the downside is "CPU-only" training. This works, but is uncomfortably slow, and should probably be replaced with a GPU-based version.

Running the actual training process

Once the training data is available, running the training process is pretty straightforward:

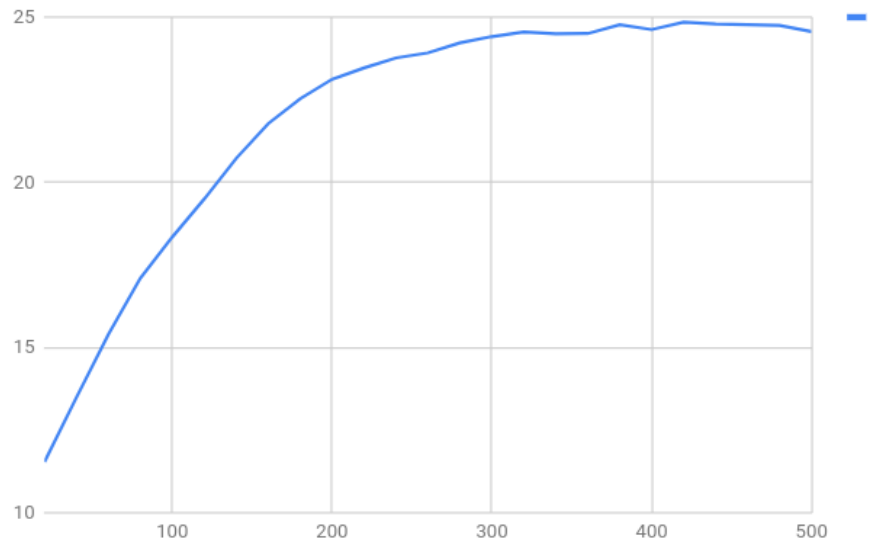
```
thomasdullien@machine-learning-training:~/sources/functionsimsearch/bin$ ./trainsimhashweights -data=/mnt/t
[!] Parsing training data.
[!] Mapping functions.txt
[!] About to count the entire feature set.
[!] Parsed 1000 lines, saw 62601 features ...
[!] Parsed 2000 lines, saw 104280 features ...
[!] Parsed 3000 lines, saw 133939 features ...
[!] Parsed 4000 lines, saw 159266 features ...
[!] Parsed 5000 lines, saw 179285 features ...
[!] Parsed 6000 lines, saw 190319 features ...
[!] Parsed 7000 lines, saw 203430 features ...
[!] Parsed 8000 lines, saw 216102 features ...
[!] Parsed 9000 lines, saw 237046 features ...
[!] Parsed 10000 lines, saw 246721 features ...
[!] Parsed 11000 lines, saw 263793 features ...
[!] Parsed 12000 lines, saw 270579 features ...
[!] Processed 12268 lines, total features are 271653
[!] Iterating over input data for the 2nd time.
[!] Loaded 12268 functions (271653 unique features)
```

```
[!] Attraction-Set: 218460 pairs
[!] Repulsion-Set: 218460 pairs
[!] Training data parsed, beginning the training process.
Itr       f       deltaf   max|g_i|   alpha      H0       rho
   0 +7.121e+04       nan 4.981e+02 2.991e-06 1.000e+00 0.000e+00
   1 +7.119e+04 2.142e+01 5.058e+02 1.000e+00 1.791e-06 3.114e-01
   2 +7.101e+04 1.792e+02 3.188e+02 1.000e+00 2.608e-05 5.735e-03
   3 +7.080e+04 2.087e+02 2.518e+02 1.000e+00 4.152e-05 4.237e-03
   4 +7.057e+04 2.271e+02 2.757e+02 1.000e+00 5.517e-05 4.469e-03
    ...
```

A few days later, the training process will have performed 500 iterations of L-BFGS while writing snapshots of the training results every 20 steps into our current directory (`20.snapshot, ..., 480.snapshot`). We can evaluate the results of our training:

```
$ for i in *.snapshot; do foo=$(./evalsimhashweights --data /mnt/june2/validation_data_seen/ --weights $i |
```

This provides us with the "difference in average distance between similar and dissimilar pairs" in the validation data: the code calculates the average distance between similar pairs and between dissimilar pairs in the validation data, and shows us the difference between the two. If our training works, the difference should go up.



We can see that somewhere around 420 training steps we begin to over-train - our difference-of-means on the validation set starts inching down again - so it is a good idea to stop the optimization process. We can also see that the difference-in-average-distance between the “similar” and “dissimilar” pairs has gone up from a bit more than 10 bits to almost 25 bits - this seems to imply that our training process is improving our ability to recognize variants of functions that we are training on.

Understanding the results of training

There are multiple ways of understanding the results of the training procedure:

Given that we can easily calculate distance matrices for a set of functions, and given that there are popular ways of visualizing high-dimensional distances (t-SNE and MDS), we can see the effects of our training visually.

Several performance metrics exist for information-retrieval tasks (Area-under-ROC-curve AUC). Nothing builds confidence like understanding, and since we obtain per-feature weights, we can manually inspect the feature weights and features to see what exactly the learning algorithm learnt.

The next sections are dedicated to go through these steps.

Using t-SNE as visualisation

A common method to visualize high-dimensional data from pairwise distances is t-SNE -- a method that ingests a matrix of distances and attempts to create a low-dimensional (2d or 3d) embedding of these points that attempts to respect distances. The code comes with a small Python script that can be used to visualize subsets of the search index.

We will create two search indices: One populated with the “learnt feature weights”, and one populated with the “unit feature weight”:

```
# Create and populate an index with the ELF unrar samples with the
# learnt features.
./createfunctionindex --index=learnt_features.index; ./growfunctionindex --index=learnt_features.index --si
# Add the PE files
for i in $(find ../testdata/PE/ -iname *.exe); do echo $i;
./addfunctionstoindex --weights=420.snapshot --index=learnt_features.index --format=PE --input=$i; done

# Create and populate an index with unit weight features.
./createfunctionindex --index=unit_features.index; ./growfunctionindex --index=unit_features.index --size_t
# Add the PE files
for i in $(find ../testdata/PE/ -iname *.exe); do echo $i;
./addfunctionstoindex --index=unit_features.index --format=PE --input=$i; done

# Dump the contents of the search index into a text file.
./dumpfunctionindex --index=learnt_features.index > learnt_index.txt
./dumpfunctionindex --index=unit_features.index > unit_index.txt

# Process the training data to create a single text file with symbols for
# all functions in the index.
cat /mnt/training_data/extracted_*.txt > ./symbols.txt

# Generate the visualisation
cd ../testdata
./plot_function_groups.py  ../bin/symbols.txt ../bin/unit_index.txt /tmp/unit_features.html
./plot_function_groups.py  ../bin/symbols.txt ../bin/learnt_index.txt /tmp/learnt_features.html
```

We now have two HTML files that use d3.js to render the results:

- Unit Feature Weights Visualisation
- Learnt Feature Weights Visualisation

Mouse-over on a point will display the function symbol and file-of-origin. It is visible to the naked eye that our training had the effect of moving groups of functions “more closely together”.

We can see here that the training does have some effect, but does not produce the same good effect for all functions: Some functions seem to benefit much more from the training than others, and it remains to be investigated why this is the case.

Using AUC for choosing a good distance threshold.

Practical searching:

Searching for unrar code in mpengine.dll

Searching for libtiff code in Adobe Acrobat Reader.