B12 The Reduction of False Negative Errors in Fuzzy Hashing

Matthew G. Nolan, BS, 1702 Byrd Street, Baltimore, MD 21230; and Michael J. Salyards, PhD*, 45 High Street, Sharpsburg, GA 30277*

After attending this presentation, attendees will understand: (1) the algorithms behind fuzzy hashing; (2) be presented the inadequacy of the false negative error rate of the original algorithm; (3) receive the statistical evidence to support the changes made; and, (4) see that the new changes do not greatly effect performance.

This presentation will impact the forensic science community by spurring discussion on error rates in digital forensics and introducing people to an improved implementation of an underused algorithm in digital forensics.

Fuzzy hashing is a hashing technique that uses a context triggered piecewise hash to compare files. The engine continuously sums bytes until the sum modulus of the block size is equal to one less than the block size. This process repeats with different block sizes until a block size is found that creates a target hash of 33-64 characters. Originally, while this is occurring, another hash is being created that is appended to the original to allow for comparisons of files with block sizes that were either 50% or 200% of the original.

This brought about a relatively serious false negative rate as the block sizes changed for specific files that were just on the edge between two different block sizes. For files of this type, fuzzy hashing is unable to match files that were ~50% of the size of the original or greater than 200% of the size of the original for certain cases. In testing, this proved to be troublesome as the file was still very similar to the original, but due to the intolerance of differences in block sizes there would be no match.

This problem would occur a relatively significant amount of the time as the length of a fuzzy hash is between 33 and 64 characters, but the distribution is uniform, meaning that these tail conditions are just as likely to occur as the more generally tolerant conditions of a hash around 48 characters which can be appended to either way and still achieve relatively good matching distances.

During testing, it was observed that the match scores reported back were generally linearly dropping to a range around 40-50 and then suddenly falling to zero. After some investigation, it was found that this had to do with the block sizes being incompatible. This generally seemed counterintuitive as a large portion of the file was still a perfect match and one would expect the match score to drop to a number where an examiner would probably need to look at the file to determine whether or not it was actually a derivative of another file.

The solution to this problem was to expand the tolerance of the hashes to block sizes. This was done by keeping a third counter which would set off to create a hash character when the rolling sum modulus four times the original block size was equal to one less than that number.

This could lead to some questioning as to how one could be sure that this modulus would be hit as the odds of a larger block size triggering a hash are considerably worse than those of a smaller block size. The principle that allows this to occur is that bytes can be treated as something close to independent uniform distributions from 0-255. The PDF of a sum of bytes is the convolution of the two uniform

distributions, which in statistical terms forms a Gaussian curve about twice the average. Some simple math can show that discrete Gaussian modulus of the block size creates a uniform distribution from 0-(block size-1), meaning that it is simply a negative binomial distribution that repeats itself each time it ends.

Knowing that the statistics approach the expected value the longer they continue, it was safe to assume that in almost all cases (for non trivial files) that the third hash would be between 8 and 16 characters which is enough information to reliably trigger a match. Producing a fourth hash though, would max out at eight characters, and since the algorithm requires more than seven consecutive characters to be the same for any match to be declared, as well as the tendency for the values that are effected in the hash to change with relatively little data added, only an exact match of the file could be found, and only if that file happened to be one of the 20% of files that happened to have an 8 byte hash.

When using this third hash appended to the first two, false error rates were on average halved, but they became much more useful in the sense of appearing more like what you would expect a human to be able to detect. It is easy for the human brain to determine visually that a file is 40% identical to another and he/she will then wonder why the hash fails to accurately classify this homology. Using the old method, files would (on average) fall off when there was a 37.5% match, but now this level of accuracy has been doubled allowing files to match down to an 18.75% match. This level of matching allows for the limitation to become the actual algorithm to match files rather than the generation of hashes.

The tradeoff for this would be an expected increase in processing time as the program is now computing three hashes instead of two. There is a very slight increase in processing time due to the fact that the extra hashes only provide new code when they are invoking their own encoding of data which is one-sixth as often as the original two pieces of the hash are invoked. This combined with the fact that there is only one extra operation per cycle which does not get invoked this allows for the increase in execution time to be only 10%.

Digital Forensics, Fuzzy Hashing, Error Rate

Copyright 2011 by the AAFS. Unless stated otherwise, noncommercial *photocopying* of editorial published in this periodical is permitted by AAFS. Permission to reprint, publish, or otherwise reproduce such material in any form other than photocopying must be obtained by AAFS. ** Presenting Author*