Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis

Fabio Pagani
EURECOM
France
fabio.pagani@eurecom.fr

Matteo Dell'Amico
Symantec Research Labs
France
matteo_dellamico@symantec.com

Davide Balzarotti
EURECOM
France
davide.balzarotti@eurecom.fr

ABSTRACT

Fuzzy hashing algorithms provide a convenient way of summarizing in a compact form the content of files, and of looking for similarities between them. Because of this, they are widely used in the security and forensics communities to look for similarities between binary program files; one version of them, ssdeep, is the de facto standard to share information about known malware.

Fuzzy hashes are quite pervasive, but no study so far answers conclusively the question of which (if any) fuzzy hashing algorithms are suited to detect similarities between programs, where we consider as similar those programs that have code or libraries in common. We measure how four popular algorithms perform in different scenarios: when they are used to correlate statically-compiled files with the libraries they use, when compiled with different flags or different compilers, and when applied to programs that share a large part of their source code. Perhaps more importantly, we provide interpretations that explain the *reasons* why results vary, sometimes widely, among apparently very similar use cases.

We find that the low-level details of the compilation process, together with the technicalities of the hashing algorithms, can explain surprising results such as similarities dropping to zero with the change of a single assembly instruction. More in general, we see that ssdeep, the de facto standard for this type of analysis, performs definitely worse than alternative algorithms; we also find that the best choice of algorithm to use varies depending on the particularities of the use case scenario.

KEYWORDS

binary analysis, fuzzy hash, malware, approximate matching

ACM Reference Format:
Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. 2018. Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis. In *CODASPY '18: Eighth ACM Conference on Data and Application Security and Privacy, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3176258.3176306

1 INTRODUCTION

Fuzzy hashes[1] were introduced in the computer security field more than a decade ago. Unlike cryptographic hashes (e.g., MD5 or SHA512), fuzzy hashes can be compared to find *similar* pieces of data. The need for comparable hashes came mainly from two different, yet related, problems. The first one, historically speaking, is spam detection: Spamsum [29] and Nilsimsa [8] compute email signatures that are compared against known unsolicited messages to label emails accordingly. The second problem comes from the forensic community, where fuzzy hashes are used to correlate forensics artifacts. In this scenario, comparable hashes can be used to locate incomplete file fragments in disk or memory dumps, or to raise red flags if files similar to known suspicious ones are present.

Fuzzy hashing is a simple and cheap solution that can be applied to arbitrary files, requires few computational resources, and produces results in a compact text format. This convenience, combined with some early promising results in binary analysis [21], is probably the main reason why ssdeep, one of the earliest fuzzy hashing algorithms, is widely adopted in the cyber security industry.

While fuzzy hashing for binary comparison is undoubtedly widespread on the industrial side, there is no academic consensus on its merits. Some works [1, 10, 19, 21, 27, 30] measure the effectiveness of different fuzzy hashing algorithms to identify similar binaries and malicious files, often with contradictory conclusions: one study may suggest that tlsh is completely ineffective for binary comparison [30] while another finds it to be one of the best available solutions for this problem [1]; these studies generally focus on understanding *if* a given algorithm works in a certain setting, but do not investigate *why*—thus missing the opportunity to fully understand this phenomenon and generalize their findings beyond the samples used in their experiments.

Other studies [6, 16, 17, 31] have instead focused on developing alternative solutions that often provide higher accuracy in exchange for a loss in convenience in terms of generality (e.g., only being applicable to binaries for a given hardware architecture, requiring dynamic analysis of the samples, or assuming each binary can be successfully disassembled) and/or needed computational and storage resources. Maybe because of these limitations, these solutions have not yet been largely adopted by the security industry.

From the existing literature, one can understand that fuzzy hashing can *sometimes* be an easy and effective solution for binary analysis problems, and yet it is unsatisfactory in many other cases; in a malware analysis scenario, fuzzy hashes seem to identify quite consistently the similarities between samples of certain malware

[1]We use the "fuzzy hashing" and "approximate matching" terms interchangeably.

families, and fail to capture any similarity for other families. Unfortunately, in a given practical setting, it is difficult to understand whether fuzzy hashing would be useful, or if one should rather need to use more sophisticated approaches.

To overcome these limitations, we propose a comprehensive analysis of how the main fuzzy hash families behave in three binary analysis case studies. Our first main contribution is to go beyond simply comparing and clustering malware samples: we discuss other common problems such as identifying statically-linked libraries, recognizing the same software across various releases, or detecting the same program across recompilation (with different flags or different compilers altogether). Our second main contribution is to avoid performing yet another large scale experiment on fuzzy hashing, which could provide useful statistics but does not allow for a manual inspection and verification of the results. Instead, we focus on few selected test cases with the goal of pinpointing the *reason* behind the individual results and provide practical examples of when each algorithm succeeds or fails at a given task.

Our experiments shed light on the low-level details that can completely overturn the similarity between binary files. For example, we found that the similarity is not just a consequence of the size of the change. So, a single assembly instruction modified at the right position in the program is sufficient to reduce the similarity to zero. Similarly, and against the common belief in the security community, it can also be enough to replace a URL with one that contains few more characters to destroy the similarity as captured by current fuzzy hash algorithms.

More in general, different algorithm families are based on distinct *concepts* of file similarity, and distinct problems are better solved with algorithms of different families. We find that CTPH—the concept behind `ssdeep`, the de-facto industry standard—is not very well suited to binary analysis in general. Depending on the problem, approaches based on n-grams (such as `tlsh`) or on statistically improbable features (such as `sdhash`) are definitely preferable.

2 BACKGROUND

Fuzzy hashes are instances of the locality-sensitive hashes (LSH) family [12]. While for traditional hashes the only meaningful operation is checking for equality, comparing LSH hashes yields non-binary similarity values, such that similar data will have similar hashes. Fuzzy hashes apply the LSH concept to arbitrary strings or files. Depending on the particular use case in which fuzzy hashes are used, different *definitions of file similarity* may be better suited. We identify three families of fuzzy hashing algorithms, which essentially differ on the concept of similarity they apply; in our experimental section, we measure how these approaches fare when they are used in the context of different binary comparison tasks.

Context-Triggered Piecewise Hashing. Context-Triggered Piecewise Hashing (CTPH) considers two files similar if they have some *identical sub-parts*. A naïve way to apply this concept would be breaking files in fixed-size blocks, hash them, and then look for collisions. Unfortunately, this simple approach is not resilient to shifts in block boundaries: for example, prepending a single byte to a file could generate different hashes for each block, hence resulting in zero similarity. This problem was first solved in the Low-Bandwidth network File System (LBFS) [20]. To find identical file chunks (and avoid unnecessary network traffic), LBFS computes the hash of n-bytes "context" sliding windows, and places block boundaries where the first m bits of the hash are zeroes: since block boundaries only depend on the surrounding n bytes, inserting or deleting short strings of bytes only changes the hashes of few file blocks, leaving the others unchanged. Rabin fingerprints [23] were used because they are efficient to compute on sliding windows, and m is the most important parameter to tune the expected block size.

In 2002 Spamsum [28] adapted the LBFS approach to classify spam. The Spamsum fuzzy hash chooses two consecutive values of m such that each file is split in around 64 blocks; for both values of m, Spamsum encodes via `base64` the least significant 6 bits of each block hash. Therefore, two spamsum hashes can be compared only if they have compatible m values (i.e., they represent files of similar size); the similarity between the two files is a function of the edit distance between the two hashes with the same m. To avoid false positives, 0 is returned if the two hashes don't have at least a 7-character substring in common. In 2006, Kornblum proposed to apply his Spamsum implementation, **`ssdeep`**, to forensics applications such as altered document and partial file matching [18]. This proposal was very successful and today `ssdeep` is used in a wide variety of cases to discover similar files.

Over the years, various improvements to `ssdeep` have been proposed. For example, `MRSH` [26] still uses CTPH to divide files, but encodes the set of hashes in Bloom filters and evaluates similarity as a function of the number of bits in common between them; this implementation obtains a better trade-off between the hash size and the quality of comparisons. **`mrsh-v2`** [2], which is evaluated experimentally in this work, is an improved version of `MRSH`. Another approach proposes Cuckoo filters [14] to further improve on hash compression at constant quality.

In general, while technical details may change, CTPH approaches are all based on the concept of recognizing identical blocks in a file. Therefore, if even small differences are widespread through the files, CPTH-based approach often fail to identify any similarity.

Statistically Improbable Features. Based on the realization that many files have frequent sequences of bytes in common (e.g., headers, zero padding, frequent words in text documents, etc.), Roussev proposed a new approach through **`sdhash`**, which looks for statistically improbable sequences of 64 bytes (*features*); a file's feature set file is represented in Bloom filters as done in `MRSH`. Since `sdhash` considers as related files with peculiar 64-byte substrings in common, it is particularly effective in finding cases where some parts are copied from the same source but it is less likely to detect files that have some sort of structural similarity but no longer strings in common (e.g., text files written in the same language).

N-Grams. Another approach to fuzzy hashing is based on the frequency distribution of n-grams (substrings of n bytes in a file), based on the idea that similar files will have similar n-gram frequency distributions. A first approach in this direction is Nilsimsa [8], which was proposed, like `spamsum`, to identify spam. Nilsimsa takes in consideration all the 5-grams in a file, and for each of them it generates the 3-grams that are sorted subsets of it (e.g., "`ace`" is generated from "`abcde`"). Each of these 3-grams is hashed into a value between 0 and 255 and an array called *accumulator* holds the counts for each of these hashes. A Nilsimsa hash is a bitmap of 256 bits

(32 bytes) where a bit is set to 1 if the corresponding value of the accumulator is larger than the median; similarity between files is computed as the number of bits in common between their bitmaps. Oliver et al. proposed **tlsh** [22], which is specifically designed for binary analysis and which we include in our experimental analysis. The main differences with Nilsimsa are that tlsh uses 2 bits per hash value of the accumulator (using quartiles rather than medians), a 1-byte checksum is used to recognize completely identical files, and the file length is encoded in the hash.

Summary. The approaches to fuzzy hashing described here define file similarity at a different and decreasing granularity. CTPH hashes blocks that are relevant subsets of the whole file; two files will result as similar if and only if they have rather large parts in common (e.g., a chapter in a text file). The statistically improbable features of sdhash are 64-byte strings; hence, sdhash will recognize files having such segments in common (e.g., a phrase in a text file). Finally, the *n*-gram approach of tlsh looks at the frequency distribution of short byte sequences; it will for example put in relations text documents that have the same or similar words, such as files written in the same language. For a more complete overview of fuzzy hashing algorithms and their applications we point the reader to the available surveys on the topic [3, 15].

3 A HISTORICAL OVERVIEW

All major fuzzy hashing algorithms were initially tested on binary files. For instance, Kornblum [18] (ssdeep) performed two simple experiments using a mutated Microsoft Word document and the header and footer of a picture in a file carving operation. Roussev [24] (sdhash) used a more comprehensive corpus with six file types (Microsoft Word and Excel, HTML, PDF, JPEG, and TXT) taken from the NPS Corpus [11]. However, none of these experiments was performed on program binaries.

In 2007, the year after ssdeep was first published, ShadowServer published the first study testing the algorithm's ability to detect similarities between malicious binaries [21]. The study included experiments with goals such as identifying packers, the same binary packed with different packers, or malware samples belonging to the same family. The authors concluded that ssdeep is neither suitable to detect similar malicious code using different packers nor to identify different programs using the same packer. However, they noticed that ssdeep can identify malware binaries belonging to the same family and therefore suggested other malware researchers and companies to share ssdeep hashes to make this comparison possible. Maybe as a result of these initial findings, over the years the security industry adopted ssdeep as the de facto standard fuzzy hash algorithm used for malware similarity. For example, today *VirusTotal*, *VirusShare*, *Malwr*, *VirusSign*, *Malware.lu*, *Malshare.com*, and *vicheck.ca* all report ssdeep hashes for their malware samples. However, the ShadowServer results were extracted from a small number of samples belonging to only two malware families. Moreover, the authors noticed that while for some binaries ssdeep performed well (even when compared across packed versions), for others the similarity was always zero. Unfortunately, they did not investigate the reason behind this phenomenon.

In 2011, Roussev published the first systematic comparison between ssdeep and his own sdhash [25]. Roussev considered three forensically sound scenarios: embedded object detection, single, and multiple common-blocks file correlation. His work also includes a second experiment using real world files from the GovDocs corpus [11]. While these experiments did not focus on program binaries or malware samples, sdhash outperformed ssdeep in all cases.

In the following year (2012), French and Casey [10] performed a first large scale evaluation of fuzzy hashing for malware analysis, using 10M PE files from the CERT Artifact Catalog. This study is particularly important because the authors look for the first time at the internal structure of Windows PE binaries to truly understand the results of the similarity. In particular, the authors distinguish changes to the code and to the data of an executable. For the code segment, the authors recognize that "*minor changes in the original source code may have unpredictably large changes in the resulting executable*" because the compilation process can permutate, reorder, and even remove pieces of code. On the other hand, for changes to the data segment the authors conclude that "*the expected changes to the resulting executable file are directly proportional to the amount of data changed. Since we only changed the values of data—not the way in they are referenced*". As we will see in Section 7, this conclusion is not correct, and even a 1-byte change in the data of a program can completely destroy the similarity between two files.

Based on their reasoning, French and Casey divided malware families in two groups: the ones whose samples only differ for their data (*generative malware*) and the ones where there is difference in the code (*evolutionary malware*). Using experiments on 1500 samples from 13 families, the authors concluded that ssdeep achieved a perfect classification for some families, but an almost negligible association for others; however, it never exhibited any false positive. On the other hand, sdhash was better at detecting files in the same family, at the price of a small false positive rate. Sadly, the authors did not examine the cases in which fuzzy hashing worked (or in which it did not) to understand if the reason was indeed the changes localized to the data or code part of the malware sample.

In 2013, Breitinger et al. [5] proposed a framework for testing fuzzy hashing algorithms. For the *sensitivity & robustness* tests, they adopted some of the experiments previously proposed by Roussev [25] as well as new ones like the random-noise-resistance test, which shows how many bytes of an input file have to be changed to result in a non-match file. The results strongly support sdhash over ssdeep. Following their framework guidelines, Breitinger and Roussev [4] tested sdhash, ssdeep, and mrsh-v2 on a real dataset of 4 457 files (unfortunately again containing no program binaries) and found that ssdeep generated less false matches, while sdhash provided a better trade-off between precision and recall.

With their proposal of tlsh in 2013, Oliver et al. evaluated their algorithm on binary files from different malware families, as well as HTML and text documents [22]. These experiments showed that tlsh consistently outperformed both ssdeep and sdhash.

In 2014, Azab et al. [1] tested four fuzzy hash algorithms according to their ability to perform malware clustering. Using two variants of the Zeus malware family, the authors noticed that both ssdeep and sdhash "*barely match binaries for the same version*". However, when used in conjunction with *k*-NN, both sdhash and tlsh provided good clustering results. Again, the authors did not discuss neither investigate why the tested algorithms provided results that were acceptable in certain settings and bad in others.

2015 was again a prolific year for the debate on the use of fuzzy hash in binary analysis. Li et al. [19] conducted another study on fuzzy hashing for malware clustering. Interestingly, they found that all algorithms perform better when applied to the sole code segment of the binary, instead of the entire file. The same paper also proposes three custom fuzzy hash algorithms that provided better clustering results than ssdeep and sdhash; unfortunately, two of them are file-dependent, and for none of them a public implementation was available at the time of writing.

Again in 2015, Soman presented a study in which ssdeep and sdhash were used to successfully correlate many APT samples [27], with zero false positives. Soman also found relationships among samples with differences in the code, disproving the common belief at the time and French and Casey's conclusions [10].

Finally, Upchurch and Zhou [30], tired of the inconsistent results reported by previous studies, introduced a manually verified, reproducible *Gold Standard Set* for malware similarity testing. On this dataset, the authors measured the performance of several similarity algorithms, including three of the ones studied in this paper: ssdeep, sdhash and tlsh. The paper highlights strong inconsistencies between the performance published in previous papers and the performances of the same algorithms on this new dataset (which are tipically much worse). While the authors say that the reason for this inconsistency is unknown, they point to dataset quality as the likely cause of the wide range of results. As we discuss in Section 7, this is indeed a fundamental problem—which however cannot be solved by having a Gold Standard Test, but by looking beyond the simple precision and recall values to understand how the similarity reported by fuzzy hash algorithm is tied to the type of malware family and the change in the PE files. Upchurch and Zhou also report two interesting findings about the algorithms tested in this work. First, they find that both ssdeep and sdhash provide better results at the minimum possible threshold (i.e., similarity larger than 0)—something that our own evaluation confirms. Second, this is the first independent study to compare tlsh with other fuzzy hashing algorithms on program binaries—and the results are exactly the opposite of what was reported by Oliver and Azab [1, 22], placing tlsh far behind ssdeep and sdhash.

Summary. The security community invested a considerable effort to understand if, and to which extent, fuzzy hash algorithms can identify similarities between program binaries. However, results were often contradictory; it seems that each algorithm can provide interesting results in some tests, while failing completely in other experiments.

French and Casey [10], in a report which was sadly often ignored by other works, got closer to the root of the problem. They understood that the compilation process can introduce unexpected changes after modifying even small parts of the program code. However, they felt that changes in the application data should not have this side-effect, and thus are more suitable to be captured by fuzzy hashes. Our results show that this is not the case.

To date, it is still not clear what users can expect from fuzzy hashes for binary analysis and classification. As a consequence, even academic papers have resorted to these techniques without fully understanding the consequences of their choice [7, 9, 13]. Our study attempts to bridge this gap through our experiments in three different case studies, as discussed in the following sections.

4 SCENARIOS AND EXPERIMENTAL SETUP

Similarity hashes are often used to identify smaller objects included in a larger one, or to detecting if two objects share a considerable part of their content. Putting these two cases in the context of binary analysis, we propose three scenarios: [I] library identification in statically linked programs, [II] comparing the same applications compiled with different toolchains and/or compiler options, and [III] comparing different versions of the same application.

Scenario I is a typical binary reverse engineering task; Scenario II is a common use case for embedded systems and firmware analysis, and Scenario III corresponds to the classic example of detecting malware evolution or samples belonging to the same malware family. We selected our scenarios and experiments with two main goals: 1) to test uses of fuzzy hashes in binary analysis beyond the traditional malware similarity problem; 2) to go beyond precision/recall and true/false positive measures, and describe for the first time the intricacies of similarity matching applied to binary files and the *reason* behind the results of the different tests.

All tests were performed on a Debian Testing box equipped with an Intel Core i7-3770, running Windows 7 in a virtual machine for the experiments on PE files. The compiler toolchains used were the most recent version available at the time of the experiments: gcc-5.4.1, gcc-6.3.0, clang-3.8.1 and icc-17.0.0. To compute fuzzy hashes we used ssdeep 2.31, sdhash 3.4, mrsh-v2 and tlsh 3.4.5, all compiled from sources.

Unlike the other algorithms, which output results between 0 (no similarity) and 100 (highest similarity), tlsh returns values that start at 0 (identical files) and grow indefinitely. To ease comparison with other algorithms, we adopt the approach of Upchurch and Zhou [30], re-normalizing tlsh scores to the $[0, 100]$ range.[2]

No packing or obfuscation was applied to the binaries used in the experiments. These transformations heavily alter the binary destroying the similarity picked up by fuzzy hashing algorithms, and we consider them outside the scope of this work.

5 SCENARIO I: LIBRARY IDENTIFICATION

The goal of this first scenario is to understand if fuzzy hashes can be used to recognize object files embedded inside a statically linked binary. This task is an application of the *embedded object detection* test proposed and already analyzed by the forensics community [25].

Since static linking is much more common in the Linux environment than in Microsoft Windows, we evaluate five popular Linux libraries (libcapstone, libevent, libjpeg, libm and libtsan). Static libraries are ar archives of relocatable object files (22 to 453 in our case). We linked each library against a small C program, obtaining five different executables.[3]

5.1 Object-to-Program Comparison

In the first test, we compute the similarity score by comparing each executable against all the individual object files contained in the static libraries. We repeated the test twice, once using the

[2]We renormalize a score x to $y = \max\{0, (300 - x)/3\}$.

[3]By default, the linker only includes files required by symbols and functions imported by the program; for this test, we forced including the object files of tested libraries.

Table 1: Library identification: true/false positive rates.

Algorithm	Entire object			.text segment		
	TP%	FP%	Err%	TP%	FP%	Err%
ssdeep	0	0	-	0	0	-
mrsh-v2	11.7	0.5	-	7.7	0.2	-
sdhash	**12.8**	**0**	-	**24.4**	**0.1**	53.9
tlsh	0.4	0.1	-	0.2	**0.1**	41.7

entire `.o` ELF files, and once matching only their `.text` segments (which does not take into consideration other sections and file headers that are not linked in the final executable). Table 1 shows the results of the two tests. We considered a successful match if `ssdeep`, `sdhash`, or `mrsh-v2` returned a similarity of at least 1 over 100 and `tlsh` returned a value lower than 300 (before the re-normalization described in Section 4).[4] The "Err" column reports instead cases in which the data was insufficient to even compute the fuzzy hash. The results were computed over 647 individual object files and false positives were computed using the same threshold, this time by matching the object files of libraries *not* linked in the executable.

These results show that not even the best algorithm in this case (`sdhash`) can link the individual object files and the corresponding statically-linked executables reliably enough. The worst performing one (`ssdeep`) *always* returned a score equal to zero, making it completely unsuitable for this scenario. In the next tests, we explore the factors that contribute to these negative results.

5.2 Impact of Library Fragmentation

In our experiments, statically-compiled binaries were larger than 1MB while the average non-empty library object file was smaller than 13KB: this difference makes the task of locating each fragment very difficult. CTPH solutions need files with comparable sizes; previous studies show that `ssdeep` works only if the embedded chunk is at least a third of the target file size [25].

Since size difference is certainly a major obstacle for this task, one may think that this problem can be mitigated by matching all object files at once, instead of one at a time. Even if the correct order is unknown, the presence of multiple fragments should improve the overall matching process - as this setup would shift the problem from the detection of a single embedded object to the easier problem of matching multiple common blocks [25].

To test if this is indeed the case, we concatenated all the library objects and all their `.text` sections in two different files, and then matched these files against the statically linked binaries. The experiment was repeated 100 times, each using a different random order of object concatenation. The best results were obtained by concatenating the full objects (probably due to strings stored in the data section). For example, in the case of `libjpeg`, fuzzy hashes were unable to match 59% of the individual object files and for the remaining `sdhash` (the best solution) gave an average score of 21. By concatenating all the object files and matching the resulting blob, `sdhash` returned 14. While this score could still be sufficient to identify the library, remember that this is a best-case scenario as all the library object files were forcefully linked to the final executable.

[4]We experimented with higher threshold values, but—confirming the findings of Upchurch and Zhou [30] discussed in Section 3—these values performed best.

To confirm whether the same result can also be obtained in a more realistic setting, we downloaded and statically compiled two real world programs, one using `libjpeg`, which had a 14 similarity score in the concatenation approach—and the other using `libevent`, which did not match at all in the same experiment. In this case, the `sdhash` similarity score decreased to 9 for `libjpeg`, while it remained zero for `libevent`.

5.3 Impact of Relocation

Relocation is the process, applied by the linker or by the dynamic loader, of adjusting addresses and offsets inside a program to reflect the actual position of symbols in memory. Static libraries contains relocatable objects, which means that the code of those object files contains several place-holders that are later filled by the linker when the file is included in the final executable.

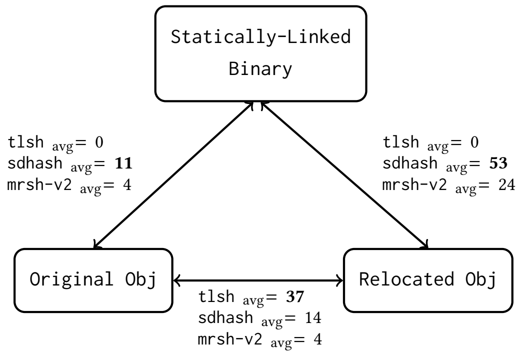

Figure 1: Average similarities after linking/relocation.

To understand the impact of relocation on similarity scores, we extracted the `.text` segments of library object files from the final binaries *after* relocations were applied by the linker. This version, which we call *relocated obj*, has the same size of the original object file, but in different bytes spread across its code a relocation was applied to a pointer. We used these files to perform two different comparisons, which are presented in Figure 1: the first is between the *relocated* and the *non relocated* versions of the same object, while the second is between the relocated object and the final executable.

On average, `sdhash` returns strong similarities between relocated objects and final binaries; this is in line with the *embedded object detection* results by Roussev [25], who showed that `sdhash` can detect files embedded in targets that are up to 10 times bigger than them. However, `sdhash` fails to recognize similarities between the relocated and not relocated versions of the same object file—thus showing that the relocation process is the main culprit for the poor results of `sdhash`. This confirms the *random-noise-resistance* test conducted by Breitinger and Bayer [5], who found that `sdhash` assigns scores greater than 10 only if less than 1% of the bytes are randomly changed. In our tests, the relocation process changes on average 10% of the object bytes, thus causing `sdhash` to fail.

Interestingly, for `tlsh` the behavior is the opposite. In fact, `tlsh` assigns high similarity to the two versions of the `.text` section (relocated and not relocated), but it is unable to match them against the final program, suggesting that in this case relocation is not the main obstacle. Figure 1 does not report results for ssdeep because it always returns a zero similarity in all tests.

Overall, we can summarize the results of the three tests we performed in this Scenario by stating that matching a static library to a statically linked binary is a difficult task, which is complicated by three main factors: 1) the fact that libraries are broken in many object files and only a subset of them is typically linked to the final executable; 2) the fact that each object file is often very small compared with the size of the statically linked binary; and 3) the fact that the content of the object files is modified (with an average byte change-rate of 10%) by the linker. Some classes of fuzzy hash algorithms are able to cope with some of these problems, but the combination of the three factors is problematic for all solutions. In fact, the *n*-gram approach of tlsh falls short when recognizing similarities between the (small) relocated object and the (large) statically-linked binary and the statistically improbable features recognized by sdhash get broken by the relocation process.

6 SCENARIO II: RE-COMPILATION

The goal of the second scenario is to recognize the same program across re-compilations—by evaluating the effect of the toolchain on the similarity scores. In particular, we look at the separate impact of the compiler and of the compilation flags used to produce the final executable. There are no previous studies about this problem, but researchers have commented that changes to the compilation process can introduce differences that hamper fuzzy hashing algorithms [10]. This scenario is also relevant to the problem of identifying vulnerable binary programs across many devices, even when libraries or software have been compiled with different options.

We test this scenario on two different sets of programs. The first one (Coreutils) contains five popular small programs (base64, cp, ls, sort, and tail) having size between 32K and 156KB each, while the second dataset (Large) contains five common larger binaries (httpd, openssl, sqlite3, ssh, and wireshark), with sizes ranging between 528KB and 7.9MB. All the experiments were repeated on four distinct versions of each program, and the results represent the average of the individual runs.

6.1 Effect of Compiler Flags

Since the number of possible flags combinations is extremely high, we limited our analysis to the sets associated to the optimization levels proposed by gcc. The first level (-O0) disables every optimization and is tipically used to ease debugging; the next three levels (-O1,-O2 and -O3) enable increasing sets of optimizations. Finally, -Os applies a subset of the -O2 flags plus other transformations to reduce the final executable size. Each test was repeated twice, once by comparing the whole binaries and once by comparing only the .text section. The first provides better results, and therefore for the sake of space we will mainly report on this case.

Results are shown in matrix plots (Figures 2 and 6–8). Histograms below the diagonal show the individual results distributions (with similarity on the X axis and percentage of the samples on the Y axis). For each algorithm, the threshold was chosen as the most conservative value that produced zero false matches. Values above the diagonal show the percentage of comparisons with a similarity greater than the threshold value. All the similarity scores used in the figure are between true positives.

We find that neither ssdeep nor its successor mrsh-v2 can reliably correlate Coreutils programs compiled at different optimization levels. However, neither algorithm ever returned a positive score when comparing unrelated binary files: hence, any result greater than zero from these tools can be considered a true match. sdhash returned low similarity scores in average (in the range 0-10) but by setting the threshold to 4 the tool generated zero false matches and was able to detect some of the utilities across all optimization levels. Finally, tlsh deserves a separate discussion. From a first glance at the matrix plot, its performance may appear very poor; this is because the graph was generated by setting the threshold at zero FP. To better understand its behavior we increased the threshold leaving 1%, 5% and 10% FP rates. Figure 6 presents the results: tlsh matches programs compiled with -O1, -O2, -O3 and -Os but cannot match programs compiled with -O0. This is reasonable as -O0 has zero optimization flags while -O1 already uses more than 50.

The picture changes slightly when testing the Large dataset programs. In this case, sdhash clearly outperforms all the other algorithms, always returning zero to unrelated files and always giving a score greater than zero to related ones.

A closer look at the data shows that all algorithms perform better using the entire file because data sections (e.g., .rodata) can remain constant when changing compiler flags. By looking at the .text section only one program was matched: openssl, which was constantly recognized also across optimization levels. We investigated this case by comparing all functions using the radiff utility, and found that many functions were unchanged even with very different compilation flags. The reason is that openssl includes many hand-written assembly routines that the compiler has to transcribe and assemble as-is, with no room for optimization.

6.2 Different Compilers

In this test we compiled each program in the Large dataset using all five optimization flags and using four different compilers: clang-3.8, gcc-5, gcc-6 and icc-17 - the Intel C compiler. The compilation process resulted in 100 distinct binaries. We then performed an all-to-all comparison with all fuzzy hash algorithms, considering as true positives the same programs compiled with a different compiler and true negatives different programs independently to the compiler used. Figure 3 summarizes the results using the matrix plot format already introduced for the previous experiment. Thresholds are again specific for each algorithm and computed to obtain a zero false positive rate.

Even if the results are better than in the previous experiment, ssdeep still performs worst. sdhash, tlsh and mrsh-v2 successfully matched all programs between gcc-5 and gcc-6 except sqlite (this is the reason why they all have 96% detection). This is because the sqlite version used (*sqlite-amalgamation*) merges the entire sqlite codebase in one single large (136k lines long) C file. This results in a single compilation unit, which gives the compiler more room to optimize (and therefore change) the code. We again show tlsh's behavior using different false positive rates in Figure 9.

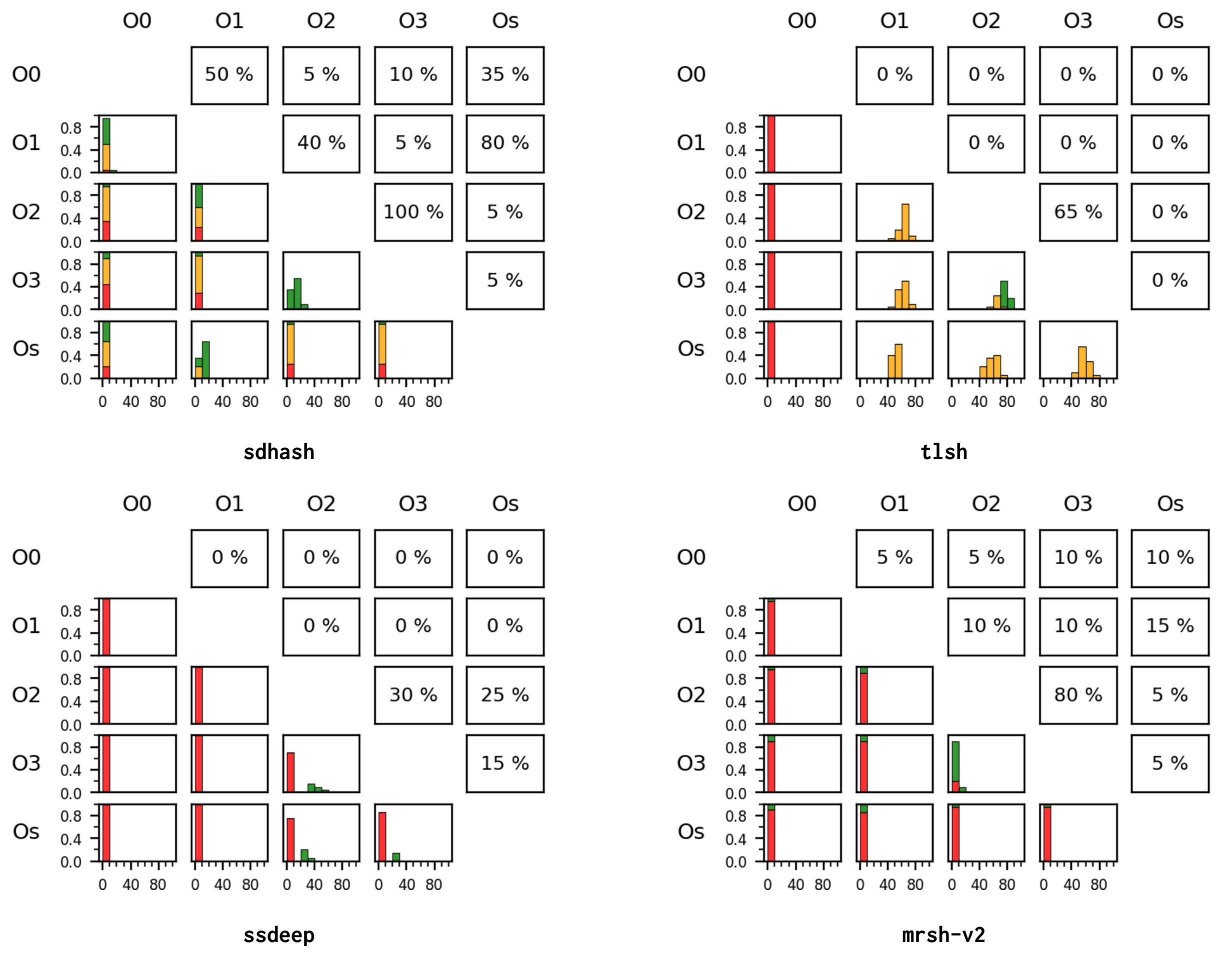

Figure 2: Coreutils compiled with different optimization levels. Red bars represent scores equal to 0, yellow bars scores below the threshold (chosen to have 0% false positive rate), green bars scores above the threshold.

7 SCENARIO III: PROGRAM SIMILARITY

Our third scenario explores one of the most interesting and common use cases for fuzzy hashing in binary analysis: the ability to correlate similar software artifacts. In our experiments we consider three types of similarity (all computed by maintaining the compilation toolchain constant): 1) binaries that originate from the same exact source code, but to whom few small changes have been applied at the assembly level; 2) binaries that originate from sources that are only different for few instructions, and 3) binaries compiled from different versions of the same software. Finally, we will compare between malware belonging to the same families to understand why fuzzy hashes work in some cases but not in others.

7.1 Small Assembly Differences

We start by assessing the impact of small-scale modifications at the assembly level, to understand their macroscopic impact on the similarity of binary programs. We apply this test to the stripped version of `ssh-client`, a medium-size program containing over 150K assembly instructions. We consider two very simple transformations: 1) randomly inserting `NOP` instructions in the binary, and 2) randomly swapping a number of instructions in the program. These transformation were implemented as target specific LLVM Pass which run very late during the compilation process. The results, obtained by repeating the experiment 100 times and averaging the similarity, are presented in Figures 4 and 5. To ease plot readability, we smoothed the curves using a moving average.

At first, the curves may seem quite counter-intuitive. In fact, the similarity seems to drop very fast (e.g., it is enough to randomly swap 10 instructions out of over 150K to drop the `sdhash` score to 38 and `ssdeep` to zero) even when more than 99.99% of the program remains unchanged. Actually, if the plots were not averaging the results over multiple runs, the picture would look much worse. For example, we observed cases in which the similarity score went to zero when just two `NOP` instructions were inserted in the binary. By manually investigating these cases, we realized that this phenomenon is due to the combination of three factors: the padding introduced by the compiler between functions, the padding added by the linker at the end of the `.text` section, and the position in which the instruction is added. In the right conditions, just few bytes are enough to increase the overall size of the code segment.

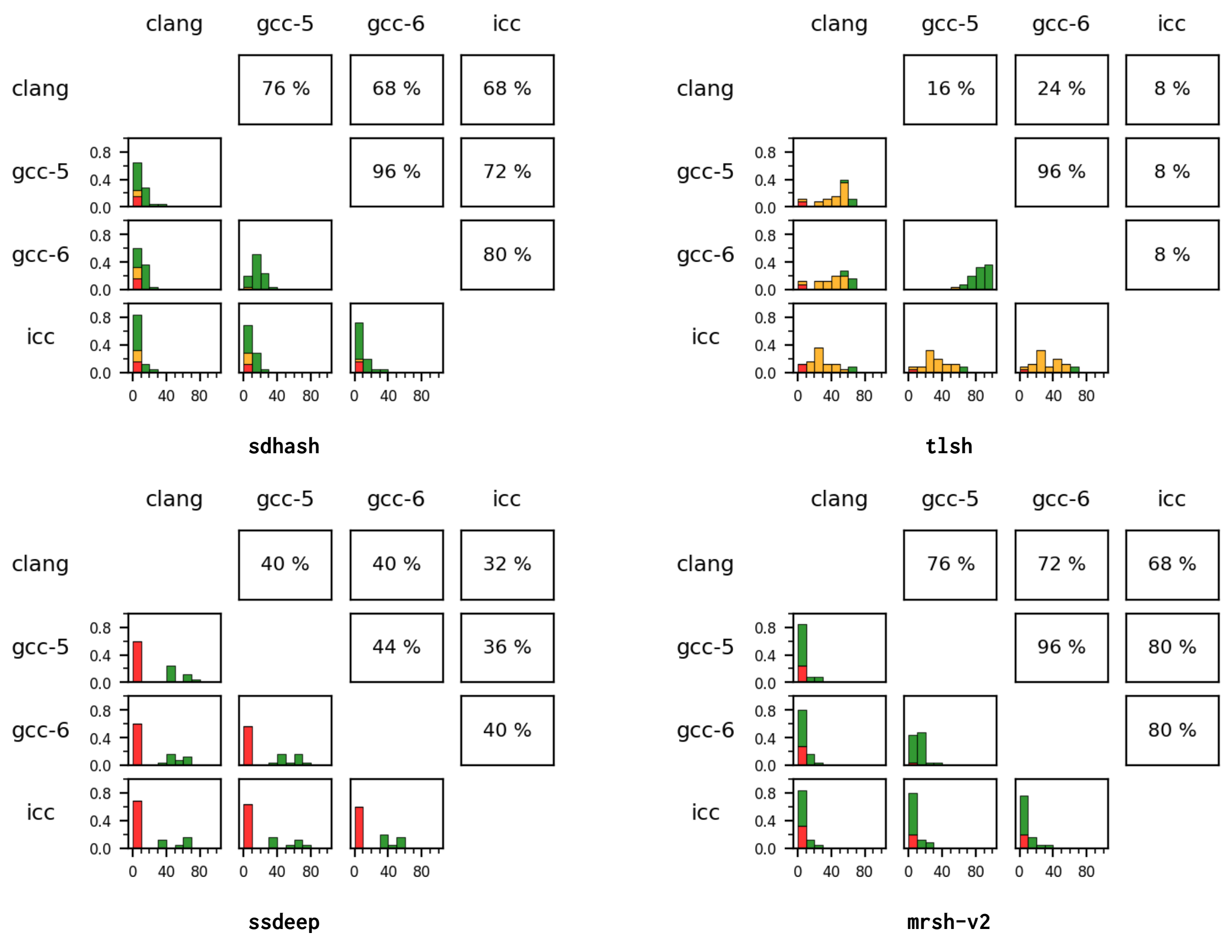

Figure 3: Large programs compiled with different compilers.

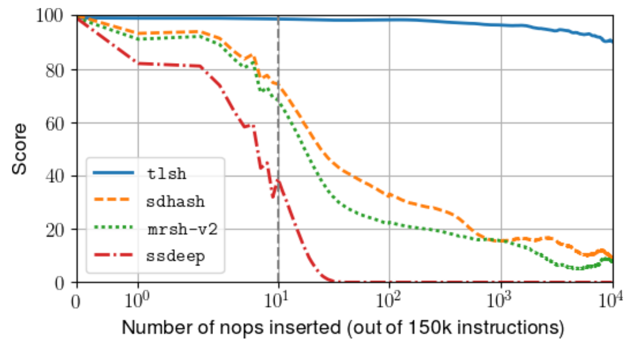

Figure 4: Inserting NOPs in random points of the program.

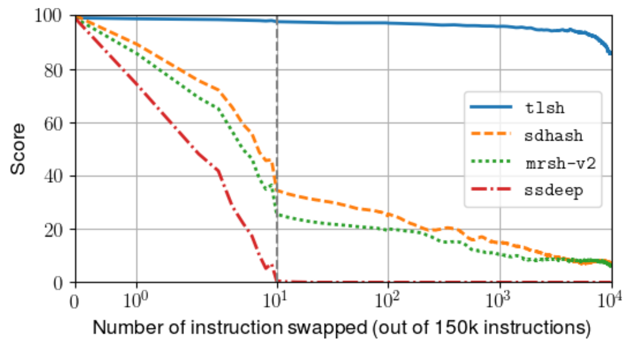

Figure 5: Swapping instructions.

As a side-effect, the successive sections are also shifted forward. The most relevant in our case is `.rodata`, which is located just after the `.text` section in memory. Shifting down this section triggers a large chain reaction in which all references in the code to global symbols are adjusted, introducing a multitude of changes spread over the `.text` section. Moreover, a substantial number of other sections needs to be adjusted: for example, consider the case where the same `.rodata` contains a jump table with relative offset to the code. Being 16 bytes farther, all these offsets needs to be adjusted as well. Another example is `.data`, which contains pointers to rodata. In total, adding two NOPs generated changes over 8 distinct sections.

To confirm this phenomenon, we wrote a linker script to increase the padding between `.text` and `.rodata`. This way, increases in the `.text` section size don't impact the position of `.rodata`. With this modification, the same two NOPs have a more limited effect, and all four algorithms are able to match the two binaries.

This effect is more pronounced in Linux than in Windows binaries. In fact, in Linux the default settings for `gcc` align both functions and sections at 16 bytes. In Windows the picture is quite different: we analyzed PE files downloaded from Internet and part of the Windows Server 2012 R2 and Windows 7 installations. While some executables don't have padding between functions, others - similarly to Linux - align functions to 16 bytes. On the other hand, the offset inside the file of every section has the lower 9 bits cleared, which effectively aligns each Section to 512 bytes. This makes Windows binaries more 'resistant' to small instructions insertion.

The only algorithm that does not seem to be affected by these microscopic changes is `tlsh`, which thanks to its frequency-based nature is able to maintain a high similarity even when 10K NOPs are inserted or instructions are swapped.

7.2 Minor Source Code Modifications

We now examine more traditional changes between programs, through small modifications to the source code of `ssh-client`. We intentionally selected changes that do not add or remove lines of code from the program:

- *Different comparison operator.* This can modify the binary in very different ways. For example, after changing a "<" to a "≤", scores remained high since the difference between binaries was just one byte. At the other end of the spectrum, compilers can statically detect that some branches are never taken and remove them from the code; in these cases, the `ssdeep` score was always 0.
- *New condition.* Again, results are quite unpredictable. Compilers can generate a few additional instructions to perform the added check, and the macroscopic effect of those instructions depends on their location and on the padding of functions and sections.
- *Change a constant value.* Effects are again hard to predict. For example, setting constants to zero can change the resulting assembly instructions (to zero a variable compilers typically emit an `xor` rather than a `mov`). Size also plays a role since the Intel instruction set offers different encodings depending on it.

Table 2: Similarity ranges for manual changes to `openssh`.

Change	`ssdeep`	`mrsh-v2`	`tlsh`	`sdhash`
Operator	0–**100**	21–**100**	**99–100**	22–**100**
Condition	0–**100**	22–99	**96**–99	37–**100**
Constant	0–97	28–99	**97**–99	35–**100**

Table 2 shows the similarity score ranges for the three experiments. Again, `tlsh` is immune to these small changes. In fact, while a single change to the source file may result in widespread changes through the binary, it is very unlikely to modify significantly the *n*-gram histogram. Other algorithms provides inconsistent results, ranging from perfect matches to poor ones (or no match at all for `ssdeep`) due to the chain effects described above.

7.3 Source Code Modifications On Malware

Our final experiment uses two real-world malware programs: Grum and Mirai. We chose those two malware families because: 1) the source code for both is available online, allowing us to compile different "variants" of each, and 2) they cover both Windows (Grum) and Linux (Mirai) environments. This experiment lets us test in a real setting the insights we gained previously.

Table 3: Manual modifications applied to malware. "M" and "G" stand respectively for "Mirai" and "Grum".

Change	`ssdeep`		`mrsh-v2`		`tlsh`		`sdhash`	
	M	G	M	G	M	G	M	G
C&C domain (real)	0	0	97	10	**99**	**88**	98	24
C&C domain (long)	0	0	44	13	**94**	**84**	72	22
Evasion	0	0	17	0	**93**	**87**	16	34
Functionality	0	0	9	0	**88**	**84**	22	7

We selected again three types of changes, reflecting differences we can expect between malware samples of the same family:

- *New C&C Domain*: we changed the Command and Control (C&C) server that the bots contact to fetch commands. Since in both cases the C&C had a default value, we decided to set it to one domain associated to the real Mirai Botnet (*real domain* in Table 3) or to an 11-character longer domain name (*long domain* in Table 3). In both malware, C&Cs are defined inside the code and not read from a configuration file, so we are sure the modification affects the compiled binary.
- *Evasion*: this modification uses real anti-analysis tricks to detect the presence of a debugger or if the sample is running inside a virtual machine. The implementation of the anti-VM function is based on timing measurements, while the anti-debugging mechanism is built around the Windows API `IsDebuggerPresent` from Grum and `ptrace` from Mirai. The cumulative sizes of these two functions is 110 bytes for Grum and 80 bytes for Mirai.
- *New Functionality*: in this case we added a malicious functionality to collect and send to the C&C the list of users present on the infected system. For Grum we used the Windows API `NetUserEnum` to retrieve the users, while for Mirai glibc's `getpwent` was used. In terms of size, Grum's functionality adds 414 bytes to the bot binary, while Mirai's one added 416 bytes.

Results are presented in Table 3. As expected, the introduction of a longer domain name had a larger impact on the similarity. In particular this is the case for Mirai, which is statically compiled. In this case, the linker combined in the final `.rodata` both the data from Mirai itself and the glibc one. The C&C string, in both experiments, is placed towards the end of the *.rodata* part reserved for Mirai, but growth is absorbed by the padding and is not enough to move the libc `.rodata`. This means that the only references that needs to be adjusted are those of Mirai, while the libc blob remains the same. The longer domain was instead larger than the padding between the "two" `.rodata`, causing the libc's `.rodata` to shift down and impact all the glibc's references.

A second observation is that in this experiment Windows variants have lower similarity than their Linux counterparts. While this seems to contradict the results presented in the previous test, the reason is that the Linux binary is statically compiled and therefore—despite the changes to the botnet code—the two binaries share big chunks of glibc code. At a closer look, we confirmed that the "Evasion" binary shares exactly the same section offsets (and thus have a good similarity score), while in the "Functionality" binaries (lower similarity) the sections are shifted down by 512 bytes.

Finally, as already shown in the previous experiments, the ability of `ssdeep` to compare binary files is very limited.

7.4 New Insights on Previous Experiments

In this final test we take a closer look at a malware dataset used in a previous study, to test if what we learned so far allows us to better understand the results of past experiments. Intrigued by the good results scored by `ssdeep` in the recent work by Upchurch and Zhou [30], and thanks to the fact that the authors shared the MD5s of the 85 samples used in their dataset, we retrieved the same set of files and use it for this final test. The samples, all manually verified by the authors [30], belong to eight different families.

We manually investigated all cases in which `ssdeep` was able to capture the similarity between files of the same family (it was the case for five of them) and found some interesting results. In two cases, the similarity was the consequence of the fact that different *dropper* files contained exactly the same second-stage program embedded in their data. In another family, 17 out of 19 files have exactly the same `.data` and `.text` sections, and 10 of them only differ in a few padding bytes at the end of the file, in what is commonly called the *file overlay*. However, since the overlay counter was zero in the PE header, it is possible that this files just contained few additional bytes because of the way they were acquired. A similar phenomenon affected also another family, in which 3 out of 6 matches were identical files except for the very last byte.

Overall, `ssdeep` was indeed able to match some different malware samples, but this happened on files that either differed only for cosmetic aspects or had large common files embedded in them.

The last and most interesting case was a family in which all similar samples had the same `rdata` section and all the section start at the same address/file offset (and therefore have the same size). In this case, a manual analysis revealed that samples shared large chunks of their code, with the only differences in the `.text` section located in parts that did not contain valid instructions (so probably related to some form of packing mechanism). In this case, due to the perfect match of the remaining part and on the fact that no offset or instruction was modified among the files, `ssdeep` successfully captured the similar malware samples.

7.5 Scenario III Findings Summary

In this third and last scenario we studied why it can be difficult to capture program modifications through fuzzy hashing algorithms. If we assume the compilation process is maintained constant among consecutive versions (we discussed its impact in the previous scenarios) we can now distinguish two cases.

If the programmer only modifies the application data (such as strings, URLs, paths and file names, global numerical constants, domain names or IP addresses) then we can expect a high similarity if the data size is not changed (e.g., if the new URL contains the exact same characters of the previous one). Otherwise, the impact depends on the number of references from the code section to data stored after the one that has been modified. In the worst case, for CTPH-based approaches and, to a minor extent, for `sdhash`, it can be enough to change few bytes in a filename to completely destroy any fuzzy hash similarity between the two versions.

It is also worth mentioning that not all "data" is part of the traditional data sections. PE files can also have overlays—i.e. bytes appended at the end of the file after all sections—and malware can embed entire files inside them. In these cases fuzzy hashes shine, as they can reliably capture the similarity between different binaries.

If the programmer modifies instead the code of the application, then the factors that affect the similarity are the size of the modification, how distributed it is on the entire code section, the function alignment used by the compiler, and the segment alignment used by the linker. Again, our experiments show that it is possible to have entirely new functions that barely modify the similarity, as well as cases in which just two `NOPs` can bring down the similarity computed by CTPH algorithms to zero.

Unlike the previous case, which was largely dominated by `sdhash`, `tlsh` shines in this scenario. In this case, the small changes applied to the assembly or source code have a very minor impact over the statistics about *n*-gram frequencies that `tlsh` uses.

8 CONCLUSIONS

This study sheds light on how fuzzy hashing algorithms behave in program analysis tasks, to help practitioners understand *if* fuzzy hashing can be used in their particular context and, if so, *which* algorithm is the best choice for the task: an important problem that is not answered conclusively by the existing literature.

Unfortunately, we found that the CTPH approach adopted by `ssdeep`—the most widely used fuzzy hashing algorithm—falls short in most tasks. We have found that other approaches (`sdhash`'s statistically improbable features and `tlsh`'s *n*-gram frequency distribution) perform way better; more in particular, we have found that `sdhash` performs well when recognizing the same program compiled in different ways, and that `tlsh` is instead very reliable in recognizing variants of the same software when the code changes.

Instead of blindly applying algorithms to recognize malware families and collecting difficult to interpret results, our evaluation looked at the details of both hashing algorithms and the compilation process: this allowed us to discover why fuzzy hashing algorithms can fail, sometimes surprisingly, in recognizing the similarity between programs that have undergone only very minor changes.

In conclusion, we show that `tlsh` and `sdhash` consistently outperform `ssdeep` and should be recommended in its place (`tlsh` is preferable when dealing with source code changes, and `sdhash` works better when changes involve the compilation toolchain); our analysis on where and why hashing algorithms are or are not effective sheds light on the impact of implementation choices, and can be used as a guideline towards the design of new algorithms.

REFERENCES

[1] Ahmad Azab, Robert Layton, Mamoun Alazab, and Jonathan Oliver. 2014. Mining malware to detect variants. In *Cybercrime and Trustworthy Computing Conference (CTC), 2014 Fifth*. IEEE, 44–53.

[2] Frank Breitinger and Harald Baier. 2012. Similarity preserving hashing: Eligible properties and a new algorithm MRSH-v2. In *International Conference on Digital Forensics and Cyber Crime*. Springer, 167–182.

[3] Frank Breitinger, Barbara Guttman, Michael McCarrin, Vassil Roussev, and Douglas White. 2014. Approximate matching: definition and terminology. *NIST Special Publication* 800 (2014), 168.

[4] Frank Breitinger and Vassil Roussev. 2014. Automated evaluation of approximate matching algorithms on real data. *Digital Investigation* 11 (2014), S10–S17.

[5] Frank Breitinger, Georgios Stivaktakis, and Harald Baier. 2013. FRASH: A framework to test algorithms of similarity hashing. In *Digital Investigation*, Vol. 10. Elsevier, S50–S58.

[6] Sagar Chaki, Cory Cohen, and Arie Gurfinkel. 2011. Supervised learning for provenance-similarity of binaries. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 15–23.

[7] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares.. In *USENIX Security*. 95–110.

[8] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. 2004. An Open Digest-based Technique for Spam Detection. *ISCA PDCS* 2004 (2004), 559–564.

[9] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, Manoj Singh Gaur, and Vijay Ganmoor. 2015. AndroSimilar: Robust signature for detecting variants of Android malware. *Journal of Information Security and Applications* 22 (2015), 66–80.

[10] David French and William Casey. 2012. Two Fuzzy Hashing Techniques in Applied Malware Analysis. *Results of SEI Line-Funded Exploratory New Starts Projects* (2012), 2.

[11] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. 2009. Bringing science to digital forensics with standardized forensic corpora. *digital investigation* 6 (2009), S2–S11.

[12] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *VLDB*. 518–529.

[13] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. 2015. Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence.. In *USENIX Security*. 1057–1072.

[14] Vikas Gupta and Frank Breitinger. 2015. How cuckoo filter can improve existing approximate matching techniques. In *International Conference on Digital Forensics and Cyber Crime*. Springer, 39–52.

[15] Vikram S Harichandran, Frank Breitinger, and Ibrahim Baggili. 2016. Bytewise Approximate Matching: The Good, The Bad, and The Unknown. *The Journal of Digital Forensics, Security and Law: JDFSL* 11, 2 (2016), 59.

[16] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 309–320.

[17] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and BS Manjunath. 2013. Sigmal: A static signal processing based malware triage. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 89–98.

[18] Jesse Kornblum. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation* 3 (2006), 91–97.

[19] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang. 2015. Experimental study of fuzzy hashing in malware clustering analysis. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*.

[20] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. 2001. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*.

[21] Digital Ninja. 2007. Fuzzy Clarity: Using Fuzzy Hashing Techniques to Identify Malicious Code. http://www.shadowserver.org/wiki/uploads/Information/FuzzyHashing.pdf. (2007).

[22] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–A Locality Sensitive Hash. In *Cybercrime and Trustworthy Computing Workshop (CTC)*.

[23] Michael O Rabin et al. 1981. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ.

[24] Vassil Roussev. 2010. Data fingerprinting with similarity digests. In *IFIP International Conference on Digital Forensics*. Springer, 207–226.

[25] Vassil Roussev. 2011. An evaluation of forensic similarity hashes. *digital investigation* 8 (2011), S34–S41.

[26] Vassil Roussev, Golden G Richard, and Lodovico Marziale. 2007. Multi-resolution similarity hashing. *digital investigation* 4 (2007), 105–113.

[27] Bhavna Soman. 2015. Ninja Correlation of APT Binaries. *First* (2015).

[28] Andrew Tridgell. 2002. spamsum. https://www.samba.org/ftp/unpacked/junkcode/spamsum/README. (2002).

[29] Andrew Tridgell. 2002. Spamsum readme. https://www.samba.org/ftp/unpacked/junkcode/spamsum/README. (2002).

[30] Jason Upchurch and Xiaobo Zhou. 2015. Variant: a malware similarity testing framework. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 31–39.

[31] Georg Wicherski. 2009. peHash: A Novel Approach to Fast Malware Clustering. *LEET* 9 (2009), 8.

9 APPENDIX

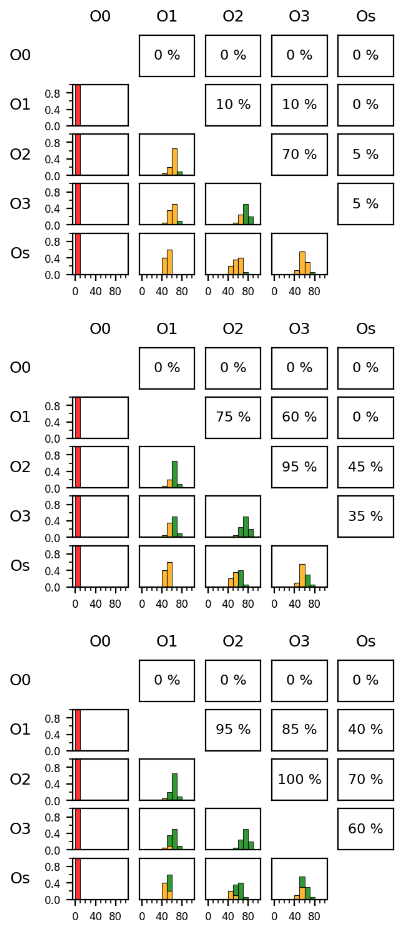

Figure 6: `tlsh` behavior on Coreutils while varying thresholds: from top to bottom, 1%, 5% and 10% false positives.

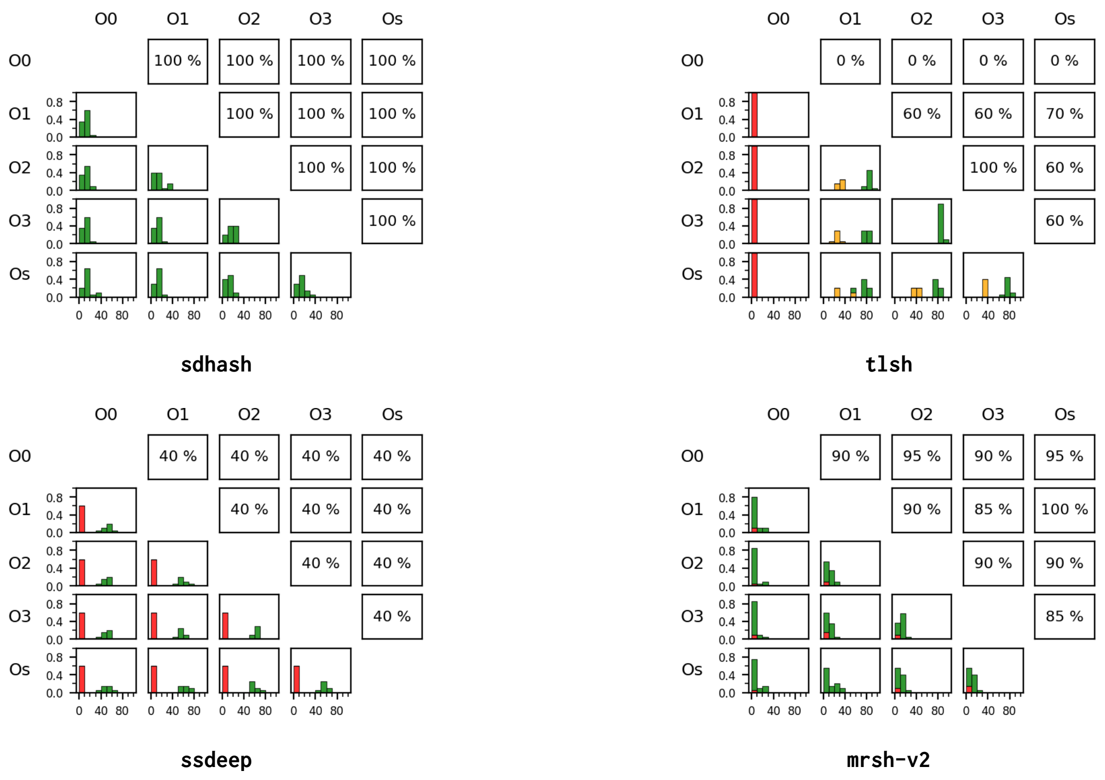

Figure 7: Programs included in the Large dataset, compiled with different optimization levels.

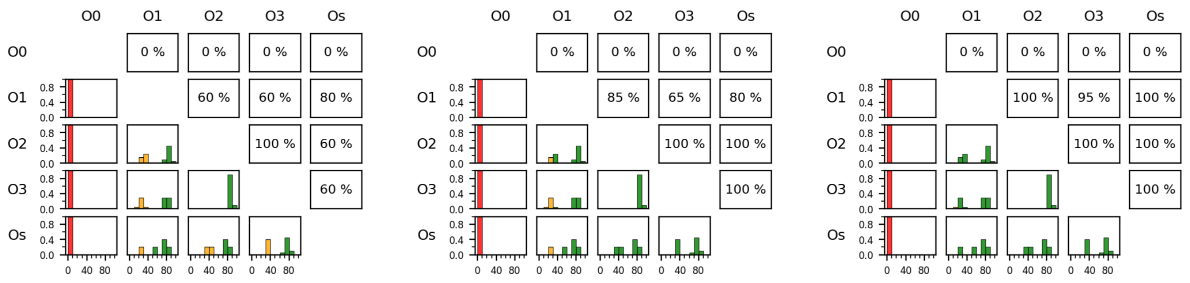

Figure 8: `tlsh` on the Large dataset, varying optimization levels and thresholds: from left to right, 1%, 5% and 10% false positives.

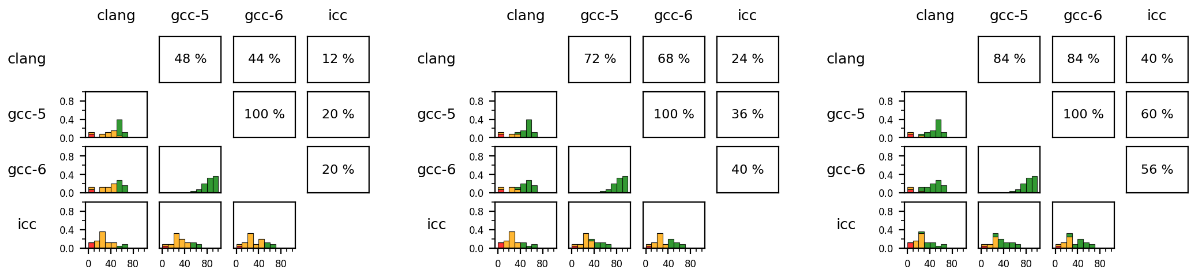

Figure 9: `tlsh` on the Large dataset, varying compilers and thresholds: from left to right, 1%, 5% and 10% false positives.