Table 1: Library identif true/false positive rates.

Algorithm	Entire			.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 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 [27].

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 fusing a different random order of object concatenation. The best results were obtained by concatenatingthe fullobjects(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, sdhashreturned 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.

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 sdhashsimilarity 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, ofadjustingaddressesandoffsetsinsideaprogramtoreflect 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.

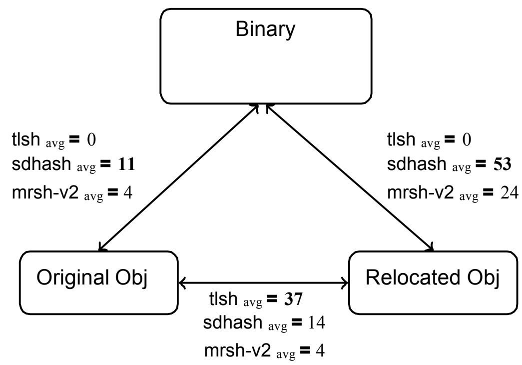


Figure 1: Average similarities after linking/relocation.

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.

0n average, sdhashreturns 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 sdhashto fail.

Interestingly, for tlshthe 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 themagainst

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

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.

1 SCENARIO II: RE-C0MPILATION

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 7.0MB. All the experiments were repeated on four distinct versions of each program, and the results represent the average of the individual runs.

1.1 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 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.

1.2 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.

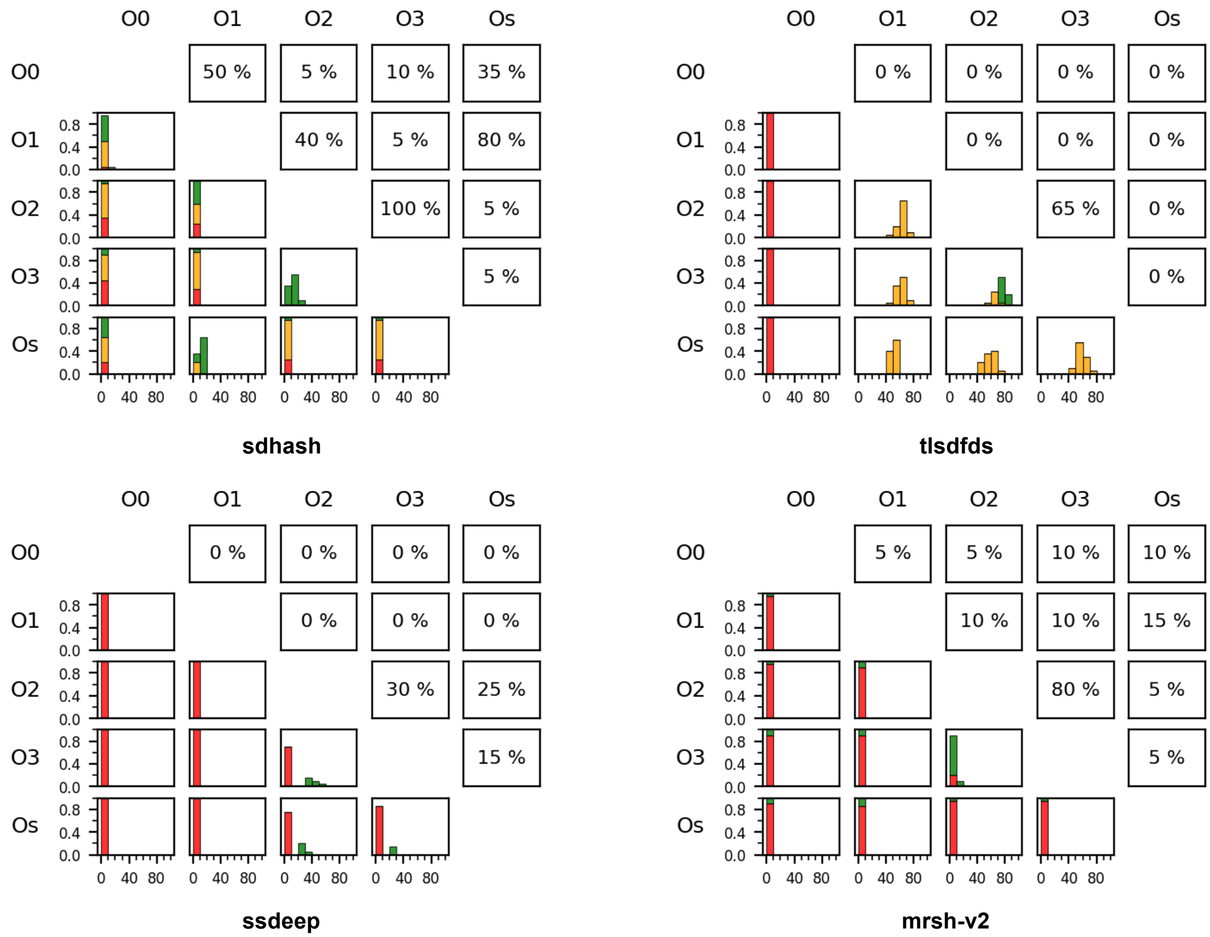


Figure 2: Coreutils compiled with different optimization levels. Red bars represent scores equal to 0, yellow bars scores below the threshold (chosen to have 0% false positive rate), green bars scores above the threshold.

2 SCENARIO III: PROGRAM SIMILARITY

Ourthirdscenarioexploresoneofthemostinterestingandcommon usecasesforfuzzyhashinginbinaryanalysis:theabilitytocorrelate 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.

2.1 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 oop 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 ssdeepto 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 .textsection, 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.

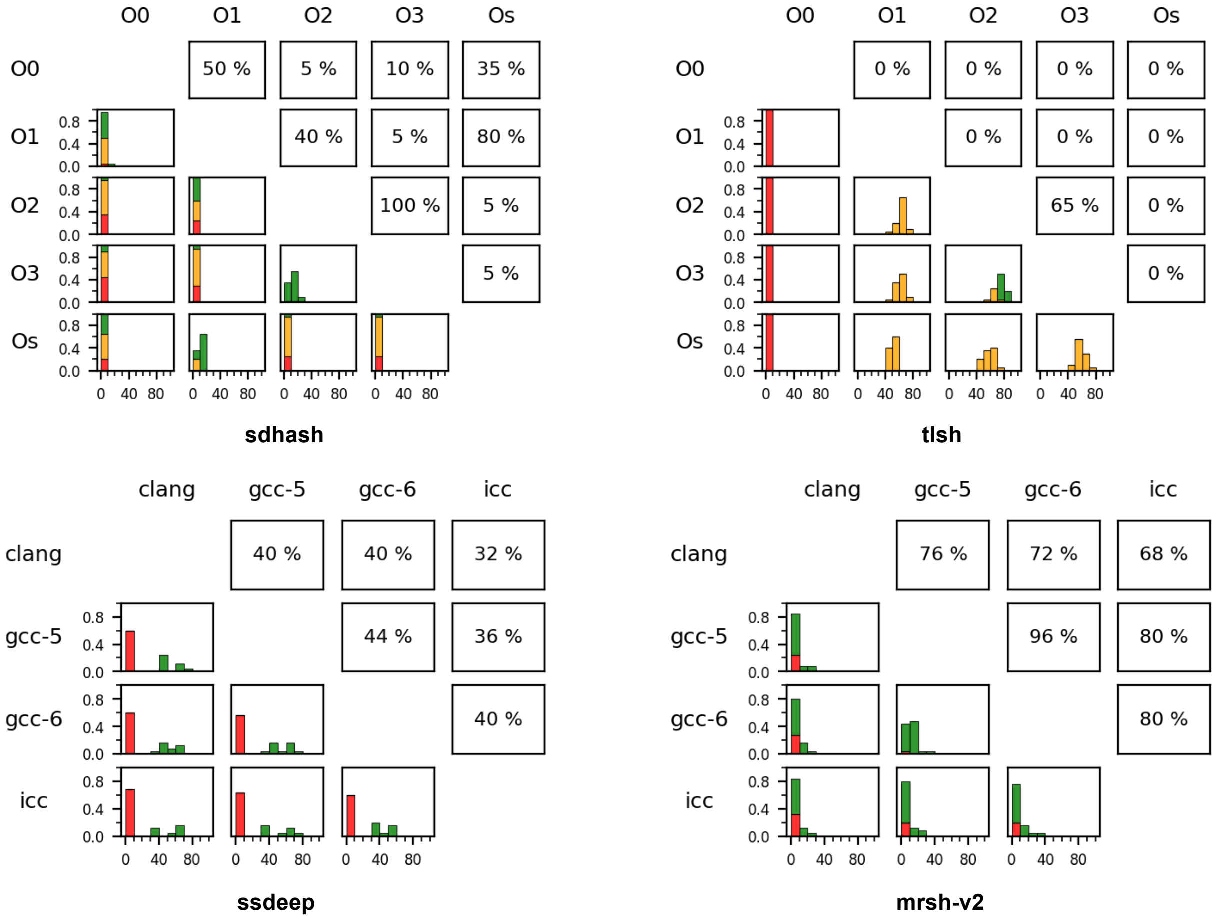


Figure 3: Large programs compiled with different compilers.

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

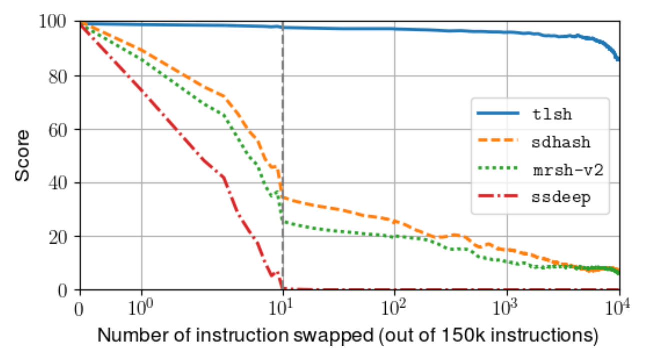


Figure 5: Swapping instructions.

As a side-effect, the successive sections are also shifted forward. The most relevant in our case is .ich is located just after the .textsection 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 casewhere 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

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

[11] files using context triggered piecewise hashing. *Digital investigation* 3 (2006), 91–97.

[12] 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)*.

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

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

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

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

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

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

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

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

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

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

[23] 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.

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

3 APPENDIX

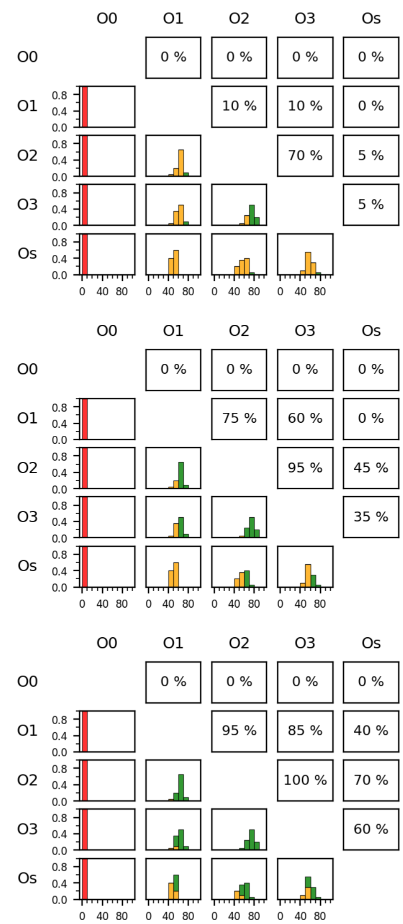
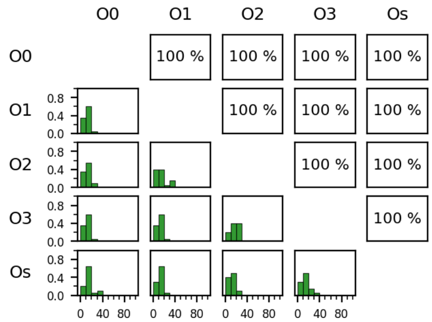
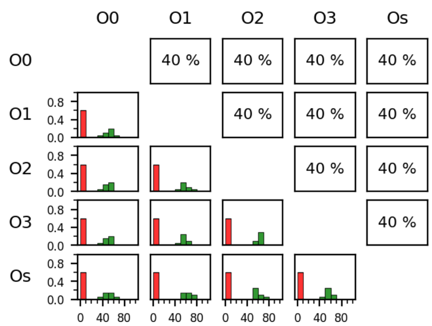


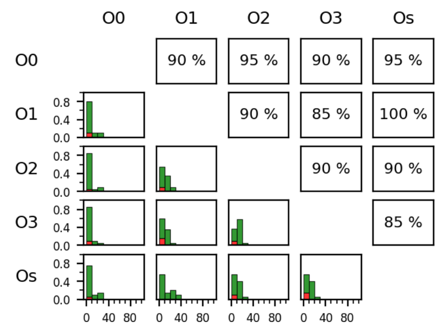
Figure 6: tlsh behaioe on Coreutils while varying thresholds: from top to bottom, 1%, 5% and 10% false positives.



sdhash

tlsh





ssdeep

mrsh-v2

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

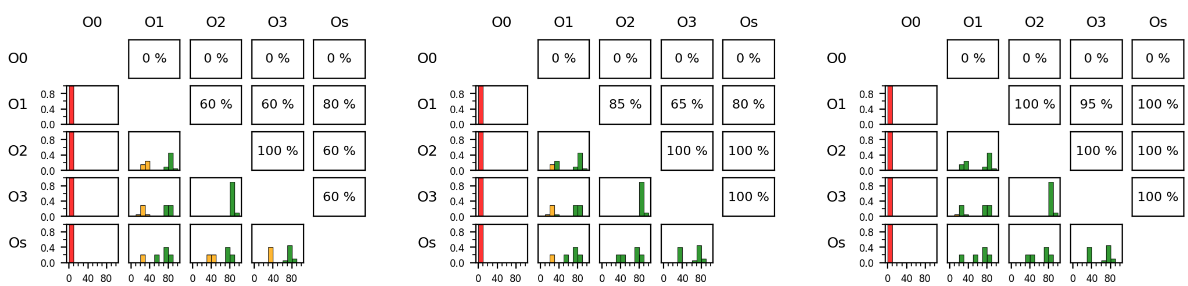


Figure 8: tlsh on the Large dataset, varying optimization levels and thresholds: from left to right, 1%, 5% and 10% false positives.

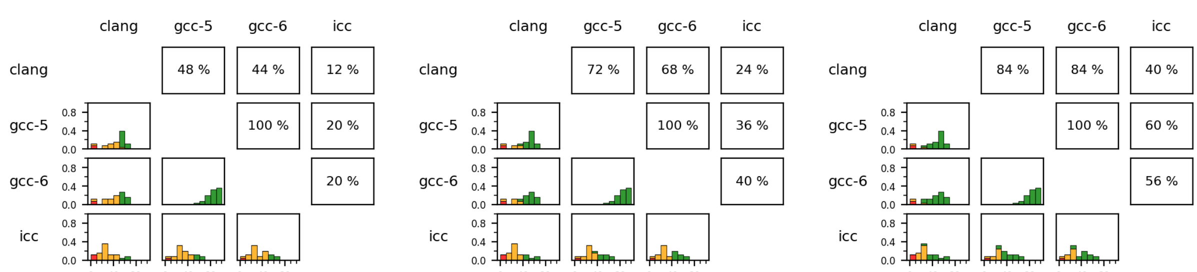


Figure 9: tlsh on the Large dataset, varying compilers and thresholds: from left to right, 1, 5% and 10% false positives.