$h(p_i)$ and $h(p_j)$, expressed as a binary vector of length R, we define the distance p_i and p_j, $D(p_i, p_j)$, to be the number of bits where $h(p_i)$ and $h(p_j)$ differ. The lower the value of the distance, the more similar the paths are. For example, a distance value of 0 means that the paths are identical, while a distance value of R means that the two paths are dissimilar. The $\mathcal{S}_{paths} = (D_{path}(p_i, p_j) : \forall(p_i, p_j) \in CFG \times \overline{CFG})$ denotes the set of pairwise comparison values (i.e., hamming distance) between paths in two distinct CFGs.

Earlier work [44] shows that one can efficiently identify whether fingerprint pairs differ in at most α bits. This value can be seen as a threshold for similarity between two fingerprints. Specifically, the lower the value of α the higher the similarity between the path blocks. Furthermore, different values of α represent different degrees of similarity. For example, $0 < \alpha < 4$ represent identical or near identical clones, while $4 < \alpha < 8$ represent similar but not near identical clones. In our approach we use $\alpha < 8$ empirically based on our experimental findings, however, other values can be used for different environment settings.

The term $\mathcal{S}_{CFG} \in [0, 1]$ computes the pairwise similarity between two CFG fingerprints. A similarity value of 1 means that the programs are similar, while a value of 0 means that the two programs CFG share no similar paths in common. Specifically, we use a variant of the Jacard index to estimate the overall similarity between the pair $(CFG, \overline{CFG})$ is then defined as:

$$\mathcal{S}_{CFG}(CFG, \overline{CFG}) = \frac{|\{s \in \mathcal{S}_{paths} : s \leq \alpha\}|}{\min(|P(CFG)|, |P(\overline{CFG})|)}. \quad (1)$$

The numerator is the number of common paths with at most α different bits and the denominator is the total number of paths in the smaller program CFG. Consequently, the overall similarity between the two programs can be seen as the degree of overall similarity between similar paths between two the programs.

One should note that the term $\mathcal{S}_{CFG}(CFG, \overline{CFG})$ computes when one CFG is contained inside of another. In other words, it considers the case where one program consists of repeated copies of another smaller program. If we want to measure the total amount of resemblance, that is proportional similarity, between two programs CFGs one can change the numerator to be the total number of common paths. However, with either measures one can use α to only consider candidates where the similarity or containment score meets a pre-determined threshold.

3. Evaluation And Discussion

In this section we present the initial results of our proof of concept along with some insights gained during our evaluation. First, we present the details of the experimental setup and the data selected for the evaluation. Then, we present the results and discuss different observations made during the experiment.

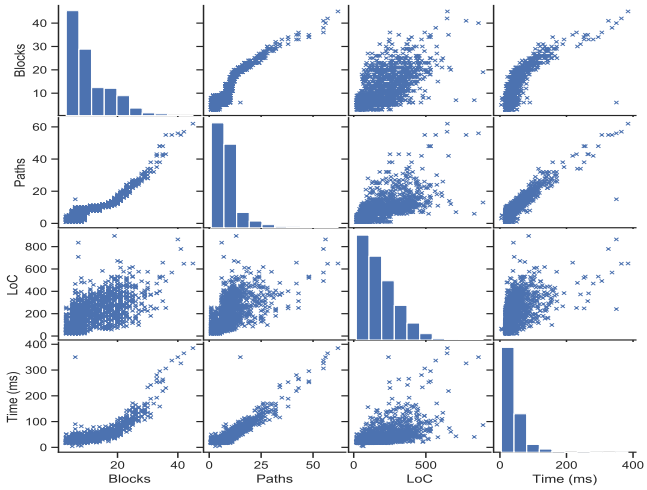

Figure 4. Pairwise plot for the experiment data set

3.1. Experiment

To evaluate the effectiveness of our approach in detecting similarities in source code, we ran the evaluation on a synthetic data set. We assessed the performance of the proposed approach against two criteria. First, execution time efficiency, and second, detection precision. The detector is implemented using Java. Specifically, the detector reads the source code files and performs normalization and generates the control flow graphs. The control flow graphs are represented using the DOT format where nodes contain the normalized source code of a block and edges connecting the blocks. All the experiments were conducted on a Windows 10 computer with an Intel Core i7 (4 cores at 3.60GHz) processor, 16GB of RAM and 1TB of HDD storage capacity.

Experimental Data. Due to the lack of a standard benchmarks for ABAP clone detection, 664 ABAP programs were collected from online repositories[1] to serve as our code base. Typically, Verifying clones is a subjective decision that depends on the analyst experience and the context of the code. Furthermore, manual verification of all code clones candidates is impractical. Therefore, We selected a 50 random programs for manual examination to evaluate the existence of false positives.

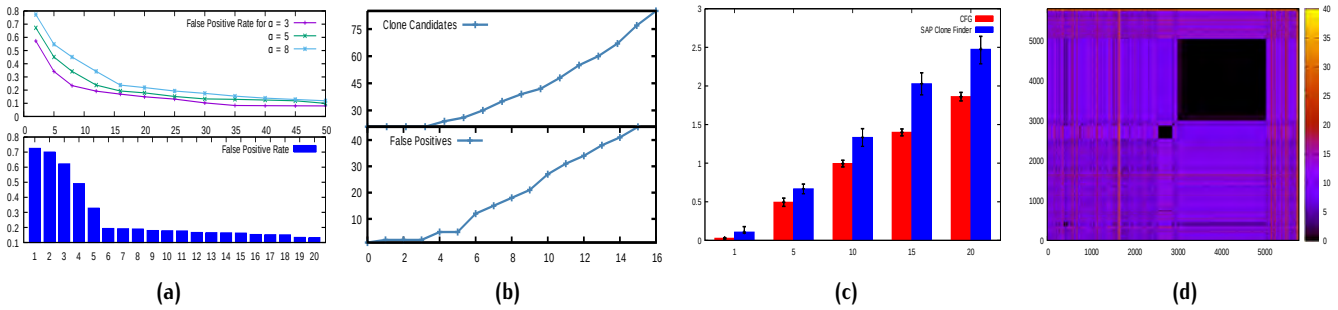

Figure 5. Detection Results: (a) False positives as a function of the number of LoC (Top), False positives as a function of the number of blocks (Bottom); (b) Clone candidates identified for α (Top), False positives for α (Bottom); (c) Time it takes to query for clones in our approach; and (d) Similarity Matrix Heat Map.

3.2. Results

Figure 4 shows the scatter plot of the source code set used in our experiments. The pairwise plots show the relationships between the number of blocks, paths, line of code, and the CFG extraction time for the programs in the data set. One can see that most of the programs in the data set contains between 5-20 blocks, that results in less than 25 paths for most of the programs in the data set. The number of blocks and paths are closely correlated, however, the number of lines in the program is less correlated to the number of path. Furthermore, the execution time grows as a function of the paths, blocks and to a lesser extent line of codes. On average the execution time was 0.17 seconds and all cases the execution time didn't exceed 0.45 seconds.

The distribution of the blocks count in programs show that a considerable number of programs with 10 or less blocks and paths with less than 3 blocks. Intuitively, programs and path with lower number of blocks are less diverse and will generate a less precise fingerprint. This can be illustrated with the help of figure 5a. The figure shows that the precision increases as the number of basic blocks increases. Furthermore, examining paths that have only 1 block revealed that most of these blocks contain generic code related to language specific style requirements such as error handling, calls, and GUI related. This code is not necessarily useful for the purposes of clone detection. Therefore, we exclude all paths that contain less than 3 blocks from our fingerprints repository.

Similarly, the precision increases as the number of lines of code in the block increases as shown figure 5a. Additionally, smaller threshold α is slightly more accurate for smaller programs, however, this improvement becomes less pronounced as the block size increases. Another observation is the the value of α is sensitive to the size of the program. For example, larger program can tolerate higher values of α with a reasonable false positives, while smaller programs will provide a more accurate results with lower values of α.

The impact of the threshold α on the number of clones candidates identified and false positives is shown in Figure 5b. The detector is able to identify clones with no false positives when $\alpha <= 4$. The false positives are still reasonable for $\alpha <= 7$, however, they become more pronounced at higher thresholds. This can be explained by the fact that at higher α means there are more bit differences between the signatures. We chose $\alpha = 5$ as reasonable threshold for our experiment. The detector is able to identify most of the relevant clones with a reasonable false positives. larger α will identify more clones candidates, however, most of the newly identified one were false positive. Smaller α only detected near-identical clones and missed many programs.

Figure 5c illustrates the performance improvements compared to the standard clone detector in SAP systems. One can notice that our detector executes in linear time and grows with the number of programs under review. Nevertheless, our detector provides statistically significant performance improvements as shown in the figure. Moreover, SAP clone finder grows at faster rate with the number of programs being converted. The number of clones being detected is also significantly higher in our approach. This can be explained by the flexibility in our fingerprint representation and the value of α.

Figure 5d shows the distance heat map (i.e., the values for the similarity matrix) for our code. The paths that are identical will have a lower distance, thus, darker colors. For example, There are quit few paths that are identical and therefore, shown in black. The structure shows that our method is able to discriminate between identical (or near identical) paths and similar paths. Furthermore, it validates our choice for α in the experiment.

[1] https://github.com/trending/abap

3.3. Discussion

While we have demonstrated the effectiveness of our approach for source code similarity search, there are still several practical issues one have to carefully consider for large scale adoption. Mainly, in performance and accuracy.

First, in very large code repository search and detection time may not be acceptable even with the improvements provided by our approach. However, one of the salient features of our CFGs fingerprint it's ability to represent group similarities. For example, one can use the fingerprints computed for each program CFG to classify the source code repository into several groups according to the similarity of the fingerprints. To find an initial match, we identify categories that are close to the fingerprint. Categories that are the farthest can then be excluded to reduce the the number of comparisons required to find an initial match. This can be done as an initial step to identify possible initial matches.

Another performance improvement is leveraging bit parallelism and bit vector arithmetic for the distance and similarity computations. For example, in [45] the Jaccard index is approximated using bit vector counting optimization. Same concept can be applied to our Similarity measure. Similarly, [44] showed how one can find fingerprints with a certain threshold α (i.e., hamming distance) efficiently.

One should also note that the accuracy and precision of our approach relies on the CFGs extraction and path enumeration techniques. Naive techniques may result in more false positives. However, more robust techniques may be computationally expensive. Therefore, they need to be considered carefully. One can leverage different optimization techniques and use self tuning methods [46, 47] to strike a balance between these conflicting requirements. Operationally, our tool may offer several static analysis techniques out of the box that may be selected and fine-tuned for the specific environment requirements.

Another challenge is the fact that inserting jumps that are never taken in the CFGs distorts the CFGs by generating paths that will never be executed. While matching based on CFGs semantics is not possible in the general case, static profiling techniques such as the ones presented here [33, 34] can be used to check for branching and path probabilities. This can improve the semantic matching accuracy of our approach.

Furthermore, the size of the CFGs is a critical factor as shown in our experiment. We excluded the smaller CFGs blocks from our signatures. We believe this is a reasonable assumption since smaller code have significantly lower chances to be copied. However, the threshold should be investigated further.

Using fine grained semantic at the block level to reveal more features is another area that is worth exploring. While some of this can be handled by fine-tuning the normalization and abstraction step in our approach, it might be helpful to explore some graph theoretic and machine learning ideas [48, 49] to further improve accuracy. This however may increase performance requirements and should be considered as an additional refining step.

4. Conclusion and Future Work

This paper demonstrated the need to efficiently identify and measure source code syntax and semantic similarities in large code base of ERP systems and presented an approach to address this need. We designed a detection approach that searches for duplicate and near duplicate code in an efficient way. By leveraging similarity hashing to concisely represent the control flow graphs of the code, the fingerprints capture the programs intrinsic characteristics. The detector uses the control flow graphs to enumerates possible execution paths and fingerprint these paths efficiently. The experiment showed the viability of our approach and illustrated how the detector can achieve reasonable accuracies efficiently compared to current tools.

While the results of presented in the evaluation look promising, they present an initial results of our ongoing research in clone detection for large systems and there is considerable work to be done. For example, in addition to continuing our empirical validation for larger and more challenging code base, we plan to continue our work in several important directions. First, the CFG extraction can be enhanced to capture exception handling, and function calls in the representation. Since SAP systems are highly integrated systems, in which data-centric programming is carried out in ABAP, one may also consider evaluating other intermediate representation such as call graphs, data flow graphs and system dependency graphs to consider the data-flow dependencies as well as control flow dependencies.

Secondly, our naive path enumeration can be improved to consider dominance relationship, loops and back edges to provide more precise paths for the CFG. It might also be worthwhile exploring the possibility of using path and execution static profiling techniques and path execution frequencies in the fingerprints to improve expressiveness of our representation.

The similarity hashing used can be also improved by exploring more intricate weights w for the most relevant parts instead of the equal weights used in the current implementation. Other similarity hashing techniques can be explored and studied. Finally, we plan to apply our approach on industrial case studies to evaluate different practical considerations, which would provide more insight into scalability and usability questions for different situations.

5. Acknowledgement

The authors would like to thank the management of Saudi Aramco for their support in publishing this article.

References

[1] Kremers M, Van Dissel H. Enterprise resource planning: ERP system migrations. Communications of the ACM. 2000;43(4):53–56.
[2] Lee J, Siau K, Hong S. Enterprise Integration with ERP and EAI. Comm of the ACM. 2003;46(2):54–60.
[3] Themistocleous M, Irani Z, O'Keefe RM, Paul R. ERP problems and application integration issues: An empirical survey. In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences. IEEE; 2001. p. 10–pp.
[4] Brehm L, Heinzl A, Markus ML. Tailoring ERP systems: a spectrum of choices and their implications. In: Proceedings of the 34th annual Hawaii international conference on system sciences. IEEE; 2001. p. 9–pp.
[5] Keller H, Thümmel WH. Official ABAP Programming Guidelines. Galileo Press; 2010.
[6] Keller H, Krüger S. ABAP objects. Sap Press; 2003.
[7] Juergens E, Deissenboeck F, Hummel B, Wagner S. Do code clones matter? In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE; 2009. p. 485–495.
[8] Gupta A, Suri B. A survey on code clone, its behavior and applications. In: Networking Communication and Data Knowledge Engineering. Springer; 2018. p. 27–39.
[9] Roy CK, Cordy JR. Benchmarks for software clone detection: A ten-year retrospective. In: 2018 IEEE 25th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE; 2018. p. 26–37.
[10] Rattan D, Bhatia R, Singh M. Software clone detection: A systematic review. Information and Software Technology. 2013;55(7):1165–1199.
[11] Tiarks R, Koschke R, Falke R. An assessment of type-3 clones as detected by state-of-the-art tools. In: Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE Inter. Working Conf. on. IEEE; 2009. p. 67–76.
[12] Juergens E, Göde N. Achieving accurate clone detection results. In: Proceedings of the 4th Inter. Workshop on Software Clones. ACM; 2010. p. 1–8.
[13] Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of computer programming. 2009;74(7):470–495.
[14] Svajlenko J, Roy CK. Evaluating modern clone detection tools. In: Software Maintenance and Evolution (ICSME), 2014 IEEE Int Conf on. IEEE; 2014. p. 321–330.
[15] Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. IEEE Trans on software eng. 2007;33(9).
[16] Thomsen MJ, Henglein F. Clone detection using rolling hashing, suffix trees and dagification: A case study. In: Software Clones (IWSC), 2012 6th Intern. Workshop on. IEEE; 2012. p. 22–28.
[17] Guo J, Zou Y. Detecting clones in business applications. In: Reverse Engineering, 2008. WCRE'08. 15th Working Conference on. IEEE; 2008. p. 91–100.
[18] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE Inter Conf on. IEEE; 1999. p. 109–118.
[19] Baker BS. On finding duplication and near-duplication in large software systems. In: Reverse Engineering, 1995., Proceedings of 2nd Working Conference on. IEEE; 1995. p. 86–95.
[20] Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. In: Software Maintenance, 1998. Proceedings., Int. Conf. on. IEEE; 1998. p. 368–377.
[21] Krinke J. Identifying similar code with program dependence graphs. In: Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. IEEE; 2001. p. 301–309.
[22] Mayrand J, Leblanc C, Merlo E. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In: icsm. vol. 96; 1996. p. 244.
[23] Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans on Software Engineering. 2002;28(7):654–670.
[24] Sheneamer A, Kalita J. Code clone detection using coarse and fine-grained hybrid approaches. In: 2015 IEEE seventh international conference on intelligent computing and information systems (ICICIS). IEEE; 2015. p. 472–480.
[25] Hummel B, Juergens E, Heinemann L, Conradt M. Index-based code clone detection: incremental, distributed, scalable. In: 2010 IEEE International Conference on Software Maintenance. IEEE; 2010. p. 1–9.
[26] Toomey W. Ctcompare: Code clone detection using hashed token sequences. In: 2012 6th Inter Workshop on Software Clones (IWSC). IEEE; 2012. p. 92–93.
[27] Uddin MS, Roy CK, Schneider KA, Hindle A. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In: Reverse Eng. (WCRE), 2011 18th Working Conf on. IEEE; 2011. p. 13–22.
[28] White M, Tufano M, Vendome C, Poshyvanyk D. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM Inter Conf on Automated Software Engineering. ACM; 2016. p. 87–98.
[29] Peng M, Xie Q, Wang H, Zhang Y, Tian G. Bayesian sparse topical coding. IEEE Transactions on Knowledge and Data Engineering. 2018;.
[30] Grove D, Chambers C. A framework for call graph construction algorithms. ACM Trans on Programming Languages and Systems (TOPLAS). 2001;23(6):685–746.
[31] Mikhailov A, Hmelnov A, Cherkashin E, Bychkov I. Control flow graph visualization in compiled software engineering. In: Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on. IEEE; 2016. p. 1313–1317.
[32] Sparks S, Embleton S, Cunningham R, Zou C. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In: Twenty-Third Annual

Computer Security Applications Conference (ACSAC 2007). IEEE; 2007. p. 477–486.
[33] Ball T, Larus JR. Efficient path profiling. In: Proc. of the 29th annual ACM/IEEE inter symposium on Microarchitecture. IEEE Computer Society; 1996. p. 46–57.
[34] Wu Y, Larus JR. Static branch frequency and program profile analysis. In: Proceedings of the 27th annual international symposium on Microarchitecture. ACM; 1994. p. 1–11.
[35] Lim HI. Comparing Control Flow Graphs of Binary Programs through Match Propagation. In: 2014 IEEE 38th Annual Computer Software and Applications Conference. IEEE; 2014. p. 598–599.
[36] Bruschi D, Martignoni L, Monga M. Detecting self-mutating malware using control-flow graph matching. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer; 2006. p. 129–143.
[37] Nandi A, Mandal A, Atreja S, Dasgupta GB, Bhattacharya S. Anomaly detection using program control flow graph mining from execution logs. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM; 2016. p. 215–224.
[38] Allen FE. Control flow analysis. In: ACM Sigplan Notices. vol. 5. ACM; 1970. p. 1–19.
[39] Wang J, Shen HT, Song J, Ji J. Hashing for similarity search: A survey. arXiv preprint arXiv:14082927. 2014;.
[40] Wang J, Zhang T, Sebe N, Shen HT, et al. A survey on learning to hash. IEEE Trans on Pattern Analysis and Machine Intelligence. 2018;40(4):769–790.
[41] Datar M, Immorlica N, Indyk P, Mirrokni VS. Locality-sensitive hashing scheme based on p-stable distributions. In: Proc. of the twentieth annual symposium on Computational geometry. ACM; 2004. p. 253–262.
[42] Charikar MS. Similarity estimation techniques from rounding algorithms. In: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. ACM; 2002. p. 380–388.
[43] Raymond JW, Willett P. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. Journal of computer-aided molecular design. 2002;16(7):521–533.
[44] Manku GS, Jain A, Das Sarma A. Detecting near-duplicates for web crawling. In: Proceedings of the 16th international conference on World Wide Web. ACM; 2007. p. 141–150.
[45] Jang J, Brumley D, Venkataraman S. Bitshred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM conference on Computer and communications security. ACM; 2011. p. 309–320.
[46] Alomari FB, Menascé DA. Self-protecting and self-optimizing database systems: Implementation and experimental evaluation. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference. ACM; 2013. p. 18.
[47] Alomari F, Menasce DA. An autonomic framework for integrating security and quality of service support in databases. In: 2012 IEEE Sixth Inter. Conf. on Software Security and Reliability. IEEE; 2012. p. 51–60.
[48] Peng M, Zhu J, Wang H, Li X, Zhang Y, Zhang X, et al. Mining event-oriented topics in microblog stream with unsupervised multi-view hierarchical embedding. ACM Transactions on Knowledge Discovery from Data (TKDD). 2018;12(3):38.
[49] Peng M, Shi H, Xie Q, Zhang Y, Wang H, Li Z, et al. Block Bayesian Sparse Topical Coding. In: 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD)). IEEE; 2018. p. 271–276.