Quantitative Symbolic Similarity Analysis*

Laboni Sarker
University of California Santa Barbara
Santa Barbara, CA, USA
labonisarker@ucsb.edu

ABSTRACT

Similarity analysis plays a crucial role in various software engineering tasks, such as detecting software changes, version merging, identifying plagiarism, and analyzing binary code. Equivalence analysis, a stricter form of similarity, focuses on determining whether different programs or versions of the same program behave identically. While extensive research exists on code and binary similarity as well as equivalence analysis, there is a lack of quantitative reasoning in these areas. Non-equivalence is a spectrum that requires deeper exploration, as it can manifest in different ways across the input domain space. This paper emphasizes the importance of quantitative reasoning on non-equivalence which arises due to semantic differences. By quantitatively reasoning about non-equivalence, it becomes possible to identify specific input ranges for which programs are equivalent or non-equivalent. We aim to address the gap in quantitative reasoning in symbolic similarity analysis, enabling a more comprehensive understanding of program behavior.

CCS CONCEPTS

• **Software and its engineering → Software verification**; **Software reliability**.

KEYWORDS

symbolic execution, equivalence, similarity, quantitative analysis, model counting

ACM Reference Format:
Laboni Sarker. 2023. Quantitative Symbolic Similarity Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3597926.3605238

1 INTRODUCTION

Similarity analysis has various applications, including detecting and understanding software changes [26], detecting source code plagiarism [25], and analyzing binary codes for tasks like patch analysis, bug search, and malware detection [18]. Equivalence analysis, a stricter form of similarity, focuses on determining if different programs or versions behave identically. It relies on techniques such as symbolic execution [21] and heuristics [3, 4, 24, 26, 29] to assess that.

```
Version 1:
double snippet(int x, int y) {
        if (x*x*x > 0){
            if(x>0 && y==10)
                return 1000;
        } else {
            if (x>0 && y==20)
                return -1000;
        }
        return 0; }
```

```
Version 2:
double snippet(int x, int y) {
        if (x*x*x > 0){
            if(y==10)//change
                return 1000;
        } else {
            if (false)//change
                return -1000;
        }
        return 0; }
```

Figure 1: Two versions of C programs of equivalent set of the dart/test from EqBench [5] benchmark

Even though there is a lot of prior work on source code or binary similarity and equivalence analysis, there is no prior work on quantitative reasoning for code similarity and equivalence. When we assert that two programs are functionally equivalent we mean that any terminating version of the programs will produce the same output for any identical input [17, 26]. On the other-hand, non-equivalence can correspond to very different scenarios:

- Two programs may produce non-identical output for all inputs. In essence, the programs maybe non-equivalent for the entire input domain. This is the most extreme case of non-equivalence.
- Another case may be, there are some inputs for which the programs generate non-identical outputs, but for the rest of the inputs, the corresponding outputs are identical.

Two programs are considered non-equivalent even when the non-equivalence arises for only one input from the whole domain. So, when we assert that two programs are non-equivalent we are not providing the full picture of how different the two programs (or two different versions of one program) are. Non-equivalence can be seen as a spectrum which can not be comprehensively reasoned about just by saying that two programs are non-equivalent. This is because, unlike equivalence, non-equivalence does not mean non-equivalence over the whole input domain. Here, we demonstrate the importance of quantitative reasoning with an example in Figure 1. The two versions of the C programs are from the EqBench [5] which is a dataset for equivalence checking. It contains 147 equivalent and 125 non-equivalent programs in both C and java languages. Both of the versions of the C programs are marked as semantically equivalent in this dataset even though they are syntactically different. However, recall that integer overflow can lead to undefined behavior in the C programs. In this case 1, the value of variable, x, gets multiplied 3 times by itself and the comparison of the resultant determines whether it will go to the else branch or not. Interestingly here, the multiplication of 3 positive, x's, can result in negative value due to integer overflow. Therefore, only classifying two programs into non-equivalent or equivalent will not show the

*This material is based on research supported by ONR Contract No. N6833523C0019, OCEANIT LABORATORIES, INC. Award #SB230168, NSF Award #2008660

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3605238

complete view on the semantic similarity of this program. We may want to know more about for which values of x and y the programs will be equivalent or non-equivalent.

After analysis, we can infer that the two program versions are equivalent with respect to C semantics only when $x \leq 1290 \wedge x \geq -1290$. If x is within that bound, then whatever the value of y is, these two programs will be equivalent. This result can only be obtained by using a more refined reasoning about non-equivalence. Note that, this information can also be used to obtain a quantitative result: the number of inputs for which the two programs are equivalent, or the percentage of the input domain for which the two programs are equivalent.

2 PROPOSED APPROACH

Given a pair of programs as inputs either binary or source code, our goal is to determine whether they are equivalent or not. If they are non-equivalent, we want to acquire further information on the input values for which the programs behave differently. To achieve our objective, we can divide our workflow into three steps.

Our first task is to collect the path constraints from the programs along with the return values using symbolic execution. All the path constraints and returns from a program will be collected and combined with disjunction operation to generate the program summary. The functional symbolic summary for Figure 1 **version 1** is: $S1 \equiv ((x \times x \times x > 0 \wedge x > 0 \wedge y = 10 \wedge return = 1000) \vee (x \times x \times x <= 0 \wedge x > 0 \wedge y = 20 \wedge return = -1000) \vee (x \times x \times x > 0 \wedge x > 0 \wedge y \neq 10 \wedge return = 0) \vee (x \times x \times x <= 0 \wedge x > 0 \wedge y \neq 20 \wedge return = 0) \vee (x \times x \times x > 0 \wedge x <= 0 \wedge return = 0) \vee (x \times x \times x <= 0 \wedge x <= 0 \wedge return = 0))$. The generated summary of **version 2** is: $S2 \equiv ((x \times x \times x > 0 \wedge y = 10 \wedge return = 1000) \vee (x \times x \times x > 0 \wedge y \neq 10 \wedge return = 0) \vee (x \times x \times x <= 0 \wedge return = 0))$.

In second step, using a constraint solver we can determine whether $S1 \Leftrightarrow S2$ holds, i.e., check equivalence. If they are equivalent, we are done. But if they are not equivalent, then we continue with further analysis in third step.

For quantitative reasoning on the non-equivalence, we first test whether the non-equivalence is true for the whole domain or not by solving the constraint $S1 \wedge S2$. If we find no solution, then we can conclude that the programs are non-equivalent for the entire input domain. If there are some solutions, then we can do more analysis. We can use model counting projected on the inputs (for Figure 1, on x and y) to find the number of solutions for which they are equivalent or non-equivalent. Then we can find the ratio of the equivalent and non-equivalent solutions with respect to the domain size. Moreover, we can also find out the inputs for which the programs act differently or similarly and by this, we have both the understanding of the input values for which the programs are equivalent (or non-equivalent) and the number of such cases.

We plan to use KLEE [8] and angr [27] for collection of path constraints and summary generation using symbolic execution on the source code and binary code, respectively. For constraint solving, we plan to use solvers like Z3 [12], ABC [1], cvc5 [7] or yices [13]. Finally, for the quantitative analysis, we can use any of the above mentioned constraint solvers with enumeration or directly use the model counting based solvers [1] for counting the number of solutions. We can also use the approximate model counting tools [9], [20] for the approximation on the number of solutions. Counting alone does not provide the input values that determine program equivalence. To address this, we can collect solutions while counting and reason about the input domain. However, exhaustive enumeration is not scalable, so heuristic-based search techniques can be employed to find solutions.

3 RELATED WORK

Binary similarity: Binary code similarity is a valuable approach used to compare and identify similarities and differences between binaries. [18] has discussed about 70 binary code similarity approaches from past 25 years and 27 of the approaches work on the semantic similarity. Three methods for finding semantic similarities include symbolic formulas, input-output pairs, and instruction/system call semantics. [16] did the basic block comparison using symbolic execution and theorem proving and used that knowledge to find the graph isomorphism for finding the overall similarity of two binaries. [30], [23] use symbolic execution on two binary paths for finding binary differences. [11] works on the statistical similarity of the binaries by decomposing procedures into small strands and calculating the similarity score of the binary accumulating the pairwise semantic matching of the strands. Function wise similarity is calculated in [14] under different environments and features using jaccard index. But no work has been done for finding similarity focusing on the input domain.

Source Code similarity and equivalence: Source code similarity is crucial for detecting source code plagiarism. A study [25] reviews plagiarism detection tools in academia, covering 150 papers. In [10], behavioral similarity approach utilizing symbolic execution [21] is utilized for plagiarism detection. Moreover, majority of the popular techniques relies on symbolic execution [21] for formally proving or refuting the equivalence of two source codes. [26] proposes differential symbolic execution where they found out about the functional difference using symbolic summaries. To improve efficiency, the study abstracted syntactically identical code segments in the compared versions and explored pre-condition, path-based differential testing. [3] proposes an alternative way than [26] for abstracting the complex code. It focuses on tracking impacted statements using static analysis but does not prune common impacted code, which can be complex and unnecessary for analysis. In contrast, [4] introduces a CEGAR-based [19] approach that abstracts complex and unnecessary code, focusing only on the statements required for establishing equivalence. [29] and [24] focus on extending symbolic equivalence checking in inter-procedural level where [29] is a modular, demand driven approach and [24] is a client-specific checker. But none of them works on quantitative reasoning.

Model counting: Quantitative program analysis is an emerging area and relies on constraint solvers for model counting. [1] has implemented an automaton-based model counting tool for string constraints, reducing the problem to path counting. [2] introduced a multi-track finite state automaton for numeric and string constraints, including combinations of both. [15] improves model counting performance using sub-formula caching. Other model counting tools include SMC [22], S3# [28], LattE [6] each targeting specific domains such as strings and linear integer arithmetic. Additionally, there is an approximation-based model counter, [9], which utilizes a hashing-based approach.

REFERENCES

[1] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. 255–272. https://doi.org/10.1007/978-3-319-21690-4_15

[2] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. Parameterized model counting for string and numeric constraints. 400–410. https://doi.org/10.1145/3236024.3236064

[3] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries, Vol. 7976. https://doi.org/10.1007/978-3-642-39176-7_7

[4] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. 13–24. https://doi.org/10.1145/3368089.3409757

[5] Sahar Badihi, Yi Li, and Julia Rubin. 2021. EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. 610–614. https://doi.org/10.1109/MSR52588.2021.00084

[6] V. Baldoni, N. Berline, J.A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu. [n. d.]. LattE integrale v1.7.2. http://www.math.ucdavis.edu/ latte/.

[7] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. *cvc5: A Versatile and Industrial-Strength SMT Solver*. 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[8] Cristian Cadar and Martin Nowack. 2021. KLEE symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer* 23 (12 2021). https://doi.org/10.1007/s10009-020-00570-3

[9] Supratik Chakraborty, Kuldeep Meel, Rakesh Mistry, and Moshe Vardi. 2015. Approximate Probabilistic Inference via Word-Level Counting. *Proceedings of the AAAI Conference on Artificial Intelligence* 30 (11 2015). https://doi.org/10.1609/aaai.v30i1.10416

[10] Hayden Cheers, Yuan Lin, and Shamus Smith. 2021. Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity. *IEEE Access* PP (03 2021), 1–1. https://doi.org/10.1109/ACCESS.2021.3069367

[11] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51 (06 2016), 266–280. https://doi.org/10.1145/2980983.2908126

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* 4963, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Bruno Dutertre and Leonardo de Moura. 2006. The Yices SMT solver. (01 2006).

[14] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components *(SEC'14)*. USENIX Association, USA, 303–317.

[15] William Eiers, Seemanta Saha, Tegan Brennan, and Tevfik Bultan. 2019. Subformula Caching for Model Counting and Quantitative Program Analysis. 453–464. https://doi.org/10.1109/ASE.2019.00050

[16] Debin Gao, Michael Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. 238–255. https://doi.org/10.1007/978-3-540-88625-9_16

[17] Benny Godlin and Ofer Strichman. 2010. Inference Rules for Proving the Equivalence of Recursive Procedures. *Acta Informatica* 45, 167–184. https://doi.org/10.1007/s00236-008-0075-2

[18] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *Comput. Surveys* 54 (04 2021), 1–38. https://doi.org/10.1145/3446371

[19] Alexey Khoroshilov, Mikhail Mandrykin, and Vadim Mutilin. 2013. Introduction to CEGAR — Counter-Example Guided Abstraction Refinement. *Proceedings of the Institute for System Programming of RAS* 24 (01 2013), 219–292. https://doi.org/10.15514/ISPRAS-2013-24-12

[20] Seonmo Kim and Stephen McCamant. 2018. SearchMC: an approximate model counter using XOR streamlining techniques. (4 2018). https://doi.org/10.6084/m9.figshare.5928604.v1

[21] James King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19 (07 1976), 385–394. https://doi.org/10.1145/360248.360252

[22] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. 2014. A Model Counter For Constraints Over Unbounded Strings. *ACM SIGPLAN Notices* 49 (06 2014). https://doi.org/10.1145/2594291.2594331

[23] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-Based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 253–270.

[24] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific equivalence checking. 441–451. https://doi.org/10.1145/3238147.3238178

[25] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Transactions on Computing Education* 19 (05 2019), 1–37. https://doi.org/10.1145/3313290

[26] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pundefinedsundefinedreanu. 2008. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 226–237. https://doi.org/10.1145/1453101.1453131

[27] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. 138–157. https://doi.org/10.1109/SP.2016.17

[28] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2017. Model Counting for Recursively-Defined Strings. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, Proceedings, Part II*. 399–418.

[29] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular Demand-Driven Analysis of Semantic Difference for Program Versions. 405–427. https://doi.org/10.1007/978-3-319-66706-5_20

[30] Shi-Chao Wang, Chu-Lei Liu, Yao Li, and Wei-Yang Xu. 2017. SemDiff: Finding Semtic Differences in Binary Programs based on Angr. *ITM Web of Conferences* 12 (01 2017), 03029. https://doi.org/10.1051/itmconf/20171203029