Abstract—Equivalence checking techniques help establish whether two versions of a program exhibit the same behavior. The majority of popular techniques for formally proving/refuting equivalence are evaluated on small and simplistic benchmarks, omitting “difficult” programming constructs, such as non-linear arithmetic, loops, floating-point arithmetic, and string and array manipulation. This hinders efficient evaluation of these techniques and the ability to establish their practical applicability in real scenarios. This paper addresses this gap by contributing EQBENCH—the largest and most comprehensive benchmark for equivalence checking analysis, which contains 147 equivalent and 125 non-equivalent cases, in both C and Java languages. We believe EQBENCH can facilitate a more realistic evaluation of equivalence checking techniques, assessing their individual strength and weaknesses. EQBENCH is publicly available at: [https://osf.io/93s5b/](https://osf.io/93s5b/).

Index Terms—Equivalence checking, benchmark, Java, C.

I. INTRODUCTION

Equivalence checking establishes whether two versions of a program have identical behavior and is used in a variety of tasks, such as verifying the correctness of software upgrades, refactoring, and optimizations [1]. The most common form of equivalence used in practice is functional equivalence (or input-output equivalence), which establishes whether two terminating versions of a program produce the same output for any identical input [2], [3]. Several equivalence checking techniques have been recently proposed [3], [4], [5], [6], [7], [8], [9]. These techniques accept as input two versions of a program and either formally prove equivalence or produce a counterexample to show that the input programs are non-equivalent.

The techniques are typically evaluated on small and simplistic benchmarks: RÊVE [10] was evaluated on 23 EQ and 9 NEQ C programs, with an average of 13.2 lines of code in each program. ModDiff [7] was evaluated on 16 EQ and 12 NEQ C programs with an average of 7.4 lines of code per program. CLEVER [8] used the ModDiff benchmark and further extended it with Python programs, bringing the total numbers to 29 EQ and 21 NEQ cases, with an average of 9.5 lines of code per program. Our earlier work, ARDIF [9], substantially extended these benchmarks, using 73 EQ and 69 NEQ programs with an average of 33.9 lines of code per program. Yet, these benchmarks are written in Java and most of them do not have a C equivalent.

The lack of a representative and common benchmark hinders the comparison between the tools. Moreover, as existing benchmarks are relatively small, that makes it difficult to realistically assess the strengths and weaknesses of existing approaches. This paper aims at addressing this gap by contributing EQBENCH—the largest and most comprehensive benchmark for equivalence checking analysis. It unifies and extends benchmarks used in earlier works [10], [7], [8], [9], contributing 147 equivalent and 125 non-equivalent cases, which we capture in both C and Java, to facilitate applicability for different equivalence checking techniques.

Moreover, a distinct feature of our benchmark is that it includes complex language constructs, such as non-linear arithmetic, loops, floating-point arithmetic, and string and array manipulations, which challenge most formal methods/equivalence checking techniques. We borrow programs containing these constructs from existing literature on evaluating formal method techniques in the presence of complex path conditions and non-linear functions [11] and adapt them to the equivalence checking context, as discussed in Section II. In comparison with our earlier ARDIF benchmark, EQBENCH includes additional cases that handle string and array manipulations, systematically collected meta-data on each benchmark which describes the types of changes it contains, and both Java and C version of each benchmark to facilitate comparison between different language-specific tools.

II. DATASET GENERATION

We now describe the process we followed for creating EQBENCH.

Data Selection. We started from including in our dataset benchmarks proposed by recent work on symbolic-execution-based equivalence checking [10], [7], [8], [9]. Besides the ARDIF benchmark [9], which we base EQBENCH on, the remaining benchmarks are relatively small and contain no complex constraints. As such, we introduced additional complexity and more diverse language constructs into the dataset. To this end, we adapted benchmarks collected by Li et al. [11] from the literature on evaluating symbolic and concolic execution methods in the presence of complex path conditions and non-linear functions [12], [13]. The methods of those benchmarks are classical numerical computation functions used in real-world distributions. For example, stat computes the mean and standard deviation of a list of numbers and tsafe is an aviation safety program that predicts and resolves the loss of separation between airplanes.
Change Injection. As benchmarks by Li et al. [11] were not originally designed for the equivalence checking problem, we injected changes in the programs to create their equivalent (EQ) and non-equivalent (NEQ) versions. Specifically, for each program, we created one EQ and one NEQ version of each of its methods with at least three lines of code (we cannot effectively inject changes in shorter methods). We then created several EQ and NEQ versions of each program: first, we created a program version per a changed method, i.e., modifying one method only and leaving the remaining ones unchanged. Then, we created one EQ version where all methods are modified to their equivalent versions and one NEQ version where all methods are modified to their non-equivalent versions. For example, for the tsafe benchmark that contains three methods, we created four EQ and four NEQ versions: three EQ + three NEQ versions for EQ or NEQ changes in one of the original methods and one EQ + one NEQ versions for EQ or NEQ changes in all methods together. Other combinations of EQ and NEQ methods are also possible, as discussed in Section IV.

When creating EQ and NEQ versions of a method, we controlled for the number, type, and location of the inserted changes. We used a random number generator to automatically pick the number of changes to inject in each method: between one and three. Given the size of the methods in our benchmark (31.4 LOC on average), we believe such a number of changes is realistic and applicable for the main equivalence checking usage scenario – regression verification, where equivalence of successive, closely related versions of a program is established.

To generate NEQ methods, we randomly selected whether to introduce each change within a loop or not. Based on that, we randomly picked an assignment or a control statement to change (we did not modify method calls, loop control statements, i.e., break and continue, etc.) Inspired by the mutation testing techniques [14] that generate faulty programs (mutants) by injecting simple syntactic changes, we inserted/deleted/updated parts of the selected statements. Specifically, we represented the selected statement as an Abstract Syntax Tree (AST) whose nodes are program variables and operators and further randomly picked an AST edit operation: insertion (Ins), deletion (Del), or update (Upd) [15].

If a selected statement was an assignment, Ins added an expression to its right-hand side, which is a set of AST nodes with a randomly selected operator (e.g., +, −, and /) and an operand of the corresponding type (e.g., Integer, Float, and String). For example, the statement c = a−b could be modified to c = a−b+10. Del removed a random set of AST nodes from the right-hand side of the statement or removed the statement altogether, in case it was not an initialization of a variable. For example, the statement int c = a+b could become int c = a or int c. Upd replaced a set of randomly-selected operators and/or operands in the right-hand side of the assignment statement by a set of new values. For example, the statement c = a−b could be modified to c = a+10 where the AST nodes − and b are replaced with + and 10, respectively.

The treatment of control statements, i.e., if and loop conditions, is similar, except that the changes were made to the control condition rather than the right-hand side of an assignment. We considered two types of control statement operators: logical (e.g., && and ||) and comparison (e.g., <= and ==) [16], randomly selecting which one to modify. Ins added random comparison and/or logical operators with the corresponding operands selected at random. For example, the statement if(a < b) could become if(a < b) && a == 0 by adding one logical and one comparison operators with the corresponding operands. Del removed random logical and/or comparison operators and, if the control condition became empty, replaced the condition by true. For example, the statement if(a < b && b == 0) could be modified to if(a < b) or to if(true). Upd replaced a set of logical and/or comparison operators and their corresponding operands at random. For example, Upd of if(a < b) could result in the statement if(a < b+10).

Table I summaries these possible changes, together with an example for each change type and the number of changes of each type that we injected during our benchmark generation process. Overall, we injected 197 changes to generate 83 NEQ methods, with 112 and 85 changes to assignment and control statements, respectively; 42% of the injected changes (83/197) were insertions, and 58% (114/197) were deletions, with 112 and 85 changes to assignment and control statements, respectively. The number of changes varied between 1 and 10, with a median of 3 changes.

To generate EQ methods that represent realistic software evolution scenarios, we applied method-level code refactoring techniques, such as, extracting and inlining variables, replacing magic numbers with variables, removing dead code, and decomposing conditionals [17], [18]. We also renamed variables and inserted dead or unreachable code [19]. Specifically, dead code is code that is executed but has no effect, e.g., adding ++ followed by −. Unreachable code is the code that is guarded by conditions that cannot hold in practice, e.g., if(false). Such redundant code, albeit not explicitly added by developers, appears in real scenarios as a result of program optimization, modification, and reuse [20]. Similar to...
TABLE II: EQ: change types and their frequencies.

<table>
<thead>
<tr>
<th>Type and Description</th>
<th>Example</th>
<th># Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refactoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extracting variables to simplify assignments</td>
<td>return q’p - p’p<em>0.05; ⇒ base=q;p; discount=p</em>0.05; return base-discount;</td>
<td>40</td>
</tr>
<tr>
<td>Inlining variables to simplify assignments</td>
<td>c=a<em>b; return c; ⇒ return a</em>b;</td>
<td>25</td>
</tr>
<tr>
<td>Replacing magic numbers with variables</td>
<td>c=a<em>b</em>9.81; ⇒ return c; ⇒ gravity=9.81; ⇒ c=a<em>b</em>gravity;</td>
<td>17</td>
</tr>
<tr>
<td>Removing dead code</td>
<td>c=a<em>b; ⇒ return a</em>a; return a*c;</td>
<td>8</td>
</tr>
<tr>
<td>Decomposing conditionals</td>
<td>if(a&gt;b &amp;&amp; c&lt;=a&lt;=c)⇒ return a; ⇒ if(inRange(a))⇒ return a+=b; return a&lt;=c;</td>
<td>5</td>
</tr>
<tr>
<td>Combining conditions having same results</td>
<td>if(a&gt;b) { return a; ⇒ if(a&gt;b)</td>
<td></td>
</tr>
<tr>
<td>Replacing nested conditions by a flat list of conditions</td>
<td>if(isDead1) { c=aa; } else if(isRetired1) { c=b; } return c; ⇒ if(isDead1) { return a; } ⇒ if(isRetired1) { return b; } return c;</td>
<td>4</td>
</tr>
<tr>
<td>Redundant code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding dead code</td>
<td>c=a;</td>
<td>63</td>
</tr>
<tr>
<td>Adding unreachable code</td>
<td>⇒ if(false) c==16;</td>
<td>5</td>
</tr>
<tr>
<td>Renaming variables</td>
<td>t=height<em>width; ⇒ area=height</em>width;</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>176</td>
</tr>
</tbody>
</table>

NEQ cases, we randomly selected whether to introduce each change within a loop or not and further randomly selected between applying refactorings, renaming variables, or inserting redundant code.

Table II shows the number of changes of each type that were introduced in EQ methods. Out of 176 changes injected to create the 83 EQ methods, 57% (101/176) changes were refactorings, 36% (63/176) were insertion of dead code, 3% (5/176) were insertion of unreachable code statements, and 4% (7/176) were variable renaming.

To validate the equivalence of the original and the produced EQ versions, an additional member of our research group independently reviewed all programs created by the first author of this paper, validating their equivalence to the original program. When possible, we also ran an equivalence checking technique to assert equivalence [9]. We further manually translated each NEQ and EQ cases from Java to C and from C to Java, as necessary.

III. DATASET CHARACTERISTICS

Dataset Statistics A summary of the benchmark is given in Table III. The first four columns of the table show the name of each benchmark, the number of EQ and NEQ versions it includes, the type of non-linear operations it contains, and the method size in lines of code (LOC) – minimum, maximum, and mean. The fifth and sixth columns show the fraction of complex non-linear arithmetic and loops in a program, averaged across EQ and NEQ versions of the benchmarks. For example, the bess benchmark contains 18 EQ and 18 NEQ versions ranging from 4 to 96 LOC, with 41.8% of complex statements on average.

Overall, considering programs from all benchmarks together, 58.8% of the programs (160 out of 272) contain at least one loop and there are 0.89 loops per program on average. Similarly, 57.3% of the programs (156 out of 272) contain at least one statement with complex non-linear arithmetic and there are 25.1% of statements with non-linear arithmetic per program on average. The last column of Table III shows the fraction of changed statements in each of the benchmarks, averaged across all EQ and NEQ versions of a benchmark. There are 14.4% changed program statements per version, on average.

Data Representation. Our benchmarks are publicly available in an archived repository, as a ZIP file [21]. For easier distribution, we also stored a version of the repository in GitHub [22]. The structure of the repository is shown in Figure 1. It is organized in 18 directories corresponding to 18 benchmarks, where each benchmark takes a row in Table III. Inside each benchmark directory, we have one folder for each program. Since we have both equivalent and non-equivalent pairs for each program, we provide two sub-folders within each program directory, i.e., EQ and NEQ. Inside each of the program pair sub-directory, there are four versions: two written in Java (oldV.java and newV.java) and two written in C (oldV.c and newV.c).

For each program version, we also provide two files that contain the meta-data describing the program, in JSON format. The files are named C-Desc.json and J-Desc.json and correspond to C and Java versions of the program, respectively. Figure 2 shows an example of the meta-data for the Java version of sign program in the airy benchmark. The file is structured as follows:

- **benchmark name**: the name of the benchmark, e.g., airy;
- **program name**: the name of the program, e.g., sign;
- **LOC**: program lines of code;
- **#loops**: the number of bounded or un-bounded loops;
- **#non-linear arithmetic**: the number of statements with non-linear arithmetic expressions;
- **changes**: description of each injected change (line number, type, and operation);
- **counter-example** (for NEQ only): values for input parameters making two programs non-equivalent.

Fig. 1: The structure of the benchmark.
TABLE III: Overview of the EqBench dataset.

<table>
<thead>
<tr>
<th>Bench.</th>
<th># EQ/NEQ Vers.</th>
<th>Operations</th>
<th>LOC</th>
<th>% Non-Linear Exp.</th>
<th># Loops</th>
<th>% Changed Stms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>REVE</td>
<td>23/9</td>
<td>Polynomials</td>
<td>16</td>
<td>0.4</td>
<td>1.3</td>
<td>9</td>
</tr>
<tr>
<td>CLEVER</td>
<td>29/21</td>
<td>Polynomials</td>
<td>4</td>
<td>0.9</td>
<td>27.3</td>
<td></td>
</tr>
<tr>
<td>airy</td>
<td>6/6</td>
<td>logarithms, arrays</td>
<td>5</td>
<td>6.5</td>
<td>0.3</td>
<td>36.5</td>
</tr>
<tr>
<td>bess</td>
<td>18/18</td>
<td>Polynomials, square roots</td>
<td>4</td>
<td>41.4</td>
<td>0.3</td>
<td>8.5</td>
</tr>
<tr>
<td>caldat</td>
<td>5/5</td>
<td>trigonometrics, strings</td>
<td>22</td>
<td>23.8</td>
<td>1.5</td>
<td>40.2</td>
</tr>
<tr>
<td>dart</td>
<td>1/1</td>
<td>Polynomials</td>
<td>10</td>
<td>9.1</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>elf</td>
<td>13/13</td>
<td>Polynomials, trigonometrics</td>
<td>6</td>
<td>37.9</td>
<td>1.5</td>
<td>7.2</td>
</tr>
<tr>
<td>frenel</td>
<td>3/3</td>
<td>Object manipulation, trigonometrics</td>
<td>62</td>
<td>100.6</td>
<td>2</td>
<td>3.4</td>
</tr>
<tr>
<td>gam</td>
<td>12/12</td>
<td>Logarithm, factorials, strings</td>
<td>7</td>
<td>32.4</td>
<td>1.4</td>
<td>10.8</td>
</tr>
<tr>
<td>pow</td>
<td>1/1</td>
<td>Exponentials</td>
<td>12</td>
<td>4.9</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>ran</td>
<td>11/11</td>
<td>Polynomials, exponentials, arrays</td>
<td>7</td>
<td>37.8</td>
<td>2.6</td>
<td>5.2</td>
</tr>
<tr>
<td>sine</td>
<td>1/1</td>
<td>Bit-vector</td>
<td>148</td>
<td>7.8</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>icas</td>
<td>4/4</td>
<td>Constant equality checks</td>
<td>11</td>
<td>26.8</td>
<td>0</td>
<td>10.4</td>
</tr>
<tr>
<td>tssle</td>
<td>4/4</td>
<td>Trigonometrics</td>
<td>9</td>
<td>31.4</td>
<td>0</td>
<td>6.2</td>
</tr>
<tr>
<td>raytrace</td>
<td>6/6</td>
<td>object manipulations, strings</td>
<td>5</td>
<td>35.1</td>
<td>0.6</td>
<td>11.2</td>
</tr>
<tr>
<td>statcalc</td>
<td>1/1</td>
<td>Exponentials</td>
<td>10</td>
<td>13.2</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>optimzation</td>
<td>3/3</td>
<td>Strings, exponentials</td>
<td>5</td>
<td>43.5</td>
<td>0</td>
<td>29.2</td>
</tr>
<tr>
<td>ej_hash</td>
<td>6/6</td>
<td>object manipulations, strings</td>
<td>5</td>
<td>28.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New</td>
<td>95/95</td>
<td>Polynomials, logarithms, Strings, etc.</td>
<td>4</td>
<td>29.3</td>
<td>0.81</td>
<td>11.9</td>
</tr>
<tr>
<td>Total</td>
<td>147/125</td>
<td></td>
<td>4</td>
<td>25.1</td>
<td>0.89</td>
<td>14.4</td>
</tr>
</tbody>
</table>

1 "benchmark name": "airy",
2 "program name": "sign",
3 "LOC": 16,
4 
5 
6 "changes": [
7 "change line": 3,
8 "change type": "Insertion",
9 "change operation": "Arithmetic"
10 ],
11 "change line": 8,
12 "change type": "Update",
13 "change operation": "Arithmetic"
14 ],
15 "change line": 11,
16 "change type": "Insertion",
17 "change operation": "Logical"
18 ],
19 "counter-example":{"a": 5.0},{"b": 5.0}]

Fig. 2: Meta-data of sign program in airy benchmark.

The C version of the file differs only in the LOC and change line fields as the changes applied to both versions are identical. The change locations in the program files (.c and .java) are also annotated by the //change comments.

IV. LIMITATIONS AND EXTENSIONS

Our main motivation for creating the EqBench is to provide evaluation target for equivalence checking techniques, allowing researchers to assess the capabilities of their tools and compare them with each other on a standard benchmark. One limitation of our benchmark is that it targets regression verification case, which assumes that substantial portions of the code is unchanged between two subsequent versions of a program. As such, our dataset might not be applicable to other applications of equivalence checking, such as verification of compiler correctness [24], optimization [25], and program synthesis [26].

Because of the manual effort needed for manually constructing and validating the EQ and NEQ cases, we created one EQ and one NEQ version for each method of the programs and a limited number of EQ and NEQ cases per program (one per each changed methods and one for all changed methods together – in both EQ and NEQ scenarios). As a straightforward extension, other combinations of changed methods, with some, but not all changed methods of both types, can easily be created to enhance the applicability of the benchmark to intra-procedural equivalence checking analysis. Another obvious way of extending our dataset is to include more equivalent/non-equivalent pairs. We invite other researchers to join our effort of extending EqBench by contributing to it directly on GitHub [22].

EqBench can be further extended by collecting examples of changes from open-source repositories. Such extension will make it possible to investigate the size and complexity of evolutionary changes, comparing them to our manually injected ones. The main challenges making such extension would be (a) dissecting a project to extract a stand-alone sample that can serve as a benchmark, (b) characterizing the change, and (c) manually labeling it as EQ or NEQ.

Another way to extend EqBench is by translating the collected cases into additional languages, e.g., Python (required by CLEVER [21]) and Boogie [27] (required by SymDiff [4]).

V. CONCLUSION

This paper contributes EqBench: a dataset of 147 equivalent and 125 non-equivalent program pairs from 18 different software projects, which we made available in both Java and C programming languages. To the best of our knowledge, our dataset is the largest, the most comprehensive, and systematically documented one. We hope it will help support equivalence checking research, by allowing researchers to assess the capabilities of their tools and compare them on a common set of programs.

Acknowledgments. Part of this work was funded by Huawei Canada and by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-040).
REFERENCES


