ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code

Sahar Badihi
University of British Columbia
Canada
shrbadihi@ece.ubc.ca

Faridah Akinotcho
University of British Columbia
Canada
fari100@ece.ubc.ca

Yi Li
Nanyang Technological University
Singapore
yi_li@ntu.edu.sg

Julia Rubin
University of British Columbia
Canada
mjulia@ece.ubc.ca

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 relies on symbolic execution – a static analysis approach that reasons about program behaviors in terms of symbolic input variables. Yet, symbolic execution is difficult to scale in practice due to complex programming constructs, such as loops and non-linear arithmetic.

This paper proposes an approach, named ARDiff, for improving the scalability of symbolic-execution-based equivalence checking techniques when comparing syntactically-similar versions of a program, e.g., for verifying the correctness of code upgrades and refactoring. Our approach relies on a set of novel heuristics to determine which parts of the versions' common code can be effectively pruned during the analysis, reducing the analysis complexity without sacrificing its effectiveness. Furthermore, we devise a new equivalence checking benchmark, extending existing benchmarks with a set of real-life methods containing complex math functions and loops. We evaluate the effectiveness and efficiency of ARDiff on this benchmark and show that it outperforms existing method-level equivalence checking techniques by solving 86% of all equivalent and 55% of non-equivalent cases, compared with 47% to 69% for equivalent and 38% to 52% for non-equivalent cases in related work.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**.

KEYWORDS

Equivalence checking, program analysis, symbolic execution.

ACM Reference Format:
Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409757

```
  bessel(norm, arg){
    acc = 200;
    res = 1;
    bess = Math.pow(2, norm) * norm;
    twoarg = 2 * arg;
    if(norm <= 0){
      res = arg *  Math.pow(2, norm);
      return res;
    }
    if(arg == 0) {
-     res = twoarg * bess;
+     res = twoarg;
    } else {
      for(j = 1; j <= norm; j--){
        bess = j * acc * twoarg;
        res = res + Math.pow(2, bess);
      }
-     res = res / (200 + bess);
+     res = res / (acc + bess);
    }
    return res;
  }
```

Figure 1: Program Versions m and m'.

1 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 optimization [30]. The most common form of equivalence used in practice is *functional equivalence*, which establishes whether two terminating versions of a program produce the same output for any identical input [25, 38].

For example, Figure 1 shows the simplified code of two consecutive versions of a method m and m' that calculates Bessel's differential equation, which is often used in scientific computing and signal processing [40]. We adopt a Git-style representation of versions [1], where lines prefixed by "+" are insertions of statements in version m' that were not present in m and lines prefixed by "-" are deletions of statements that were present in version m. Despite syntactical differences, these two versions are functionally

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409757

equivalent and the goal of our work is to prove equivalence or non-equivalence of such cases.

Symbolic execution [28] – a static program analysis technique that uses symbolic rather than concrete values to represent program inputs – is one popular approach for establishing functional equivalence. With symbolic execution, a program is represented by a first-order logic formula over symbolic variables, which captures each program execution path as a conjunction between *path condition* and *path effect*: the first is the formula constraining values that enable the executing of the path and the second describes values computed along the path. This representation is referred to as a program *symbolic summary*. Establishing functional equivalence between two versions of a program then translates into the problem of establishing equivalence between their summaries, usually using solvers for Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) [14, 18, 38].

The limitations of symbolic execution are well-known: unbounded loops and recursions lead to a large number of paths and the exact number of iterations is difficult/impossible to determine statically. For the example in Figure 1, the number of iterations of the *for* loop in lines 13-16 depends on the exact value of the input parameter *norm*, which is unknown during static analysis. Thus, most techniques rely on a user-specified *bound* for the number of iterations, e.g., two or five. Bounded symbolic execution is not as complete as the unbounded one and might miss feasible behaviors, e.g., in the *sixth* execution of the loop. Furthermore, complex expressions in symbolic summaries, such as non-linear integer arithmetic, might lead to expressions in summaries that are intractable for modern decision procedures, which produce an 'unknown' result in these cases [15, 47]. An example of such non-linear arithmetic is the power operation in lines 4 and 15 of Figure 1.

In the context of functional equivalence checking, two main approaches for dealing with these limitations of symbolic execution have been proposed: Differential Symbolic Execution (DSE) [38] uses *uninterpreted functions* – function symbols that abstract internal code representation and only guarantee returning the same value given the same input parameters – to abstract syntactically identical segments of the code in the compared versions, e.g., the code in lines 2-9 as well as in lines 13-16 in Figure 1. The main idea behind this approach is to hide the complexity of this common code and skip executing it symbolically. That is, while the "naïve" approach for constructing symbolic summaries of the methods m and m' in Figure 1 would unwind the loop in lines 13-16 up to a user-specified bound, the execution of this loop can be skipped altogether as it does not affect the equivalence of these methods: the values computed in the loop are used in the same manner in both symbolic summaries of versions m and m'. Even in the presence of uninterpreted functions, the equivalence of these summaries can be established by an SMT solver using the theory of equality and uninterpreted functions [8, 29].

However, abstracting all syntactically identical code is not effective: first, it can abstract away important information needed to establish equivalence. For example, the code *acc = 200* in line 2, albeit common to both methods, is essential for establishing equivalence of statements *res = res / (200 + bess)* and *res = res / (acc + bess)* in line 17. That is, an over-approximation introduced by abstraction may lead to spurious false-negative results, characterizing programs as not equivalent when they are, in fact, equivalent. That happens because values assumed for uninterpreted functions may not correspond to real code behavior. Second, introducing numerous uninterpreted functions, with complex relationships between them, may result in unnecessary complexity in symbolic summaries, producing 'unknown' results that could be eliminated when the abstracted away code is relatively simple and is used "as is".

The regression verification using impacted summaries (IMP-S) approach [6] proposes an alternative way to abstract complex code. Instead of identifying common code blocks, it uses static analysis to identify all statements impacted by the changed code. The tool then prunes all parts of symbolic summaries that do not contain any impacted statements. The authors of IMP-S formally prove that the results produced by such an approach are correct for a given loop bound, i.e., if the approach determines the compared programs equivalent (non-equivalent) they indeed are equivalent (non-equivalent) for that bound. However, this approach assumes that all impacted statements, including common statements with complex logic, can affect the decision about the equivalence of two programs. This assumption results in inclusion of unnecessary statements and constraints in the symbolic summaries, leading to 'unknown' results that could otherwise be eliminated. Another weakness of this approach is that the conservative nature of static analysis may mark certain unimpacted statements as impacted, further inflating the produced summaries.

ModDiff [45] and CLEVER [34] focus on extending equivalence analysis to work in an inter-procedural manner. While these techniques also perform some pruning of common code, they only do that at a path level, eliminating full path summaries only if the entire path contains no changed statement. At a method-level, these techniques thus suffer from the same and even more severe limitation than IMP-S.

To summarize, the techniques proposed by existing work suffer from false-negatives and are unable to eliminate a large portion of undecidable cases, i.e., cases when an SAT/SMT solver returns 'unknown' as the result. That is because they are either too conservative and abstract larger portions of the program than actually needed to establish equivalence (DSE) or leave unnecessary parts of the program in the summaries and prevent the solver from successfully resolving these cases (IMP-S, ModDiff, CLEVER).

In this paper, we propose a method level analysis for abstracting a *portion* of code statements, aiming to arrive at an optimal abstraction which hides complex statements not necessary for establishing equivalence while keeping statements needed to prove equivalence. The goal of our work is to decrease the number of 'unknown' results and increase the number of provable results (without loop bounding) compared with earlier work. Inspired by the Counterexample-Guided Abstraction Refinement (CEGAR) paradigm [13], our approach, implemented in a tool named ARDiff, conservatively abstracts all unchanged blocks, as done by DSE, and then iteratively refines these abstracted blocks, guided by a set of heuristics for identifying the most prominent refinement candidates. Unlike prior work that mostly focuses on reducing the size of symbolic summaries, treating their complexity as a side effect, our approach balances the size, the complexity, and the expressiveness of the summaries, aiming at producing concise yet solvable expressions.

To evaluate the efficiency and effectiveness of ARDiff, we extend the set of benchmarks collected by Li et al. [32], which were inspired by classical numerical computation functions and were used for evaluating symbolic execution methods in the presence of complex path conditions and non-linear functions [19, 44]. For example, the *bess* benchmark used as the baseline for our motivating example in Figure 1 contains 17 methods used to compute Bessel's differential equation. Another benchmark, *tsafe*, contains three methods borrowed from an aviation safety program that predicts and resolves the loss of separation between airplanes.

We adapted these benchmarks for the equivalence checking context by systematically injecting realistic changes into each of the 57 benchmark's methods, producing one equivalent and one non-equivalent version of each method. We opted for using benchmarks by Li et al., in addition to those introduced by Trostanetski et al. [34, 45], for our evaluation because the latter benchmarks are relatively small and contain no complex constraints.

We compare the efficiency and effectiveness of ARDiff for establishing method-level equivalence with that of existing work: DSE and IMP-S. Our evaluation results show that ARDiff is able to establish equivalence in 63 out of 73 cases (86%) and non-equivalence in 38 out of 69 cases (55%). For equivalent cases, this is substantially higher than the results produced by IMP-S and DSE: 51 and 35 cases, respectively. For non-equivalent cases, ARDiff performs comparably and even slightly better than other tools: it is able to solve 37 out of 69 non-equivalent cases while IMP-S solves 36 and DSE – 27 cases.

Contributions. This paper makes the following contributions:

(1) It introduces a CEGAR-like abstraction/refinement approach that uses uninterpreted functions to abstract a large portion of common code and employs a number of heuristics helping to refine only abstractions that are needed to determine equivalence.

(2) It provides the first publicly-available implementation of DSE and IMP-S, as well as our novel approach named ARDiff, all in a generic framework for determining method-level functional equivalence.

(3) It introduces a non-trivial benchmark for method-level functional equivalence checking, with 57 samples of equivalent and non-equivalent method pairs. The samples of the benchmark include loop and complex non-linear arithmetic.

(4) It empirically demonstrates the effectiveness and efficiency of ARDiff compared with DSE and IMP-S.

Our implementation of ARDiff, DSE, and IMP-S, as well as our experimental data, are available online [5].

Organization. The remainder of the paper is structured as follows. Section 2 provides the necessary background and definitions used for the rest of the paper. We discuss existing techniques and outline their limitations that motivated our work in Section 3. We describe the main idea behind ARDiff in Section 4 and its implementation in Section 4.4. Section 5 describes our evaluation methodology, including benchmark construction, and the evaluation results. We discuss the related work in Section 6 and conclude the paper in Section 7 with a summary and suggestions for future research.

2 BACKGROUND

In this section, we provide the necessary background on program analysis and equivalence checking that will be used in the remainder of the paper.

Programs. We formalize the ideas in the paper in the context of a simple imperative programming language where all operations are either assignments or method calls and all variables range over integers and doubles. We assume that each program method m performs a transformation on the values of the input parameters and returns a set of values. Without loss of generality, we represent m's printing statements as return values and also assume that global variables can be passed as input parameters and return values along each path of the method. We assume that methods have no additional side-effects. We also assume that all executions of m terminate, but this assumption does not prevent m from possibly having an infinite number of paths, such as in the case where there is a loop whose number of iterations depends on an input variable.

Control and Data Dependencies. For two statements s_1 and s_2, we say that s_2 is *control-dependent* on s_1 if, during execution, s_1 can directly affect whether s_2 is executed [23]. For the example in Figure 1, statements in lines 2-6, 10, and 19 are control-dependent on the method definition in line 1. Statements inside the *for* loop in lines 14 and 15 are control-dependent on the loop declaration in line 13 which, in turn, is control-dependent on the *if* statement in line 10.

We say that statement s_2 is *data-dependent* on statement s_1 if s_1 sets a value for a variable and s_2 uses that value. For the example in Figure 1, the statement in line 8 is data-dependent on the statement in line 7 because it uses the value of the variable *res* set in line 7.

Symbolic Summaries. Symbolic execution [28] is a program analysis technique for evaluating the behavior of a program on all possible inputs. It starts by assigning symbolic values to all input parameters. For the example in Figure 1, we denote the two symbolic inputs corresponding to input parameters *norm* and *arg* (line 1) by N and A. It then executes a program with symbolic rather than concrete inputs.

A *symbolic summary* for a method m is a first-order formula M over a set of input parameters and output variables. To build a symbolic summary, the symbolic execution technique systematically explores all *execution paths* of a method, maintaining a *symbolic state* for each possible path. The symbolic state consists of two parts: *path condition* – a first-order formula that describes the conditions satisfied by the branches taken along that path, and *effect* – a mapping of program variables to expressions calculating their values in terms of symbolic inputs.

To collect all paths, when a conditional statement, such as *if* or *for*, is reached during the symbolic execution, the symbolic state of the explored path is cloned and two paths are created: in one the path condition is conjuncted with the predicate of the condition and in the other – with its negation; symbolic execution then continues to explore both paths independently. For non-conditional statements, such as assignments, it extends the symbolic state with a new expression that associates the variable on the left-hand side of the assignment with the symbolic expression for calculating its value. For example, the condition on the path spanning the lines

1-9 in both versions of the method in Figure 1 is $N \leq 0$ and the effect of the path is *Ret=A*Math.pow(2,N)*, where *Ret* represents the output variable and *Math.pow* represents the power operation from non-linear arithmetic. The effect is calculated as a multiplication of *arg* and *Math.pow(2,N)* (line 7).

The exact number of loop iterations can depend on values of input variables, which are unknown statically, e.g., in the *for* loop in lines 13-16 of Figure 1. To compute the symbolic summary, the loops are thus *bounded* to a particular user-defined value. With a bound of 2, the loop in our example induces two paths: with one and with two iterations over the loop. Skipping the loop altogether (zero iterations) is impossible in this program because the loop is reachable only if the value of *norm* is greater than 0 (see lines 6-9). With loop bounding, symbolic execution has the potential to under-approximate the program's behaviors, e.g., those that happen in subsequent iterations of the loop.

A symbolic summary of a path is a conjunction of its path condition and symbolic state, e.g., $N \leq 0 \wedge Ret = A * Math.pow(2, N)$. The symbolic summary of a method is a disjunction of symbolic summaries of all its paths. E.g., the symbolic summary of the method m in Figure 1, with the loop bound of 2, is:

$$(N \leq 0 \wedge Ret = A * Math.pow(2, N)) \vee$$
$$(N > 0 \wedge A = 0 \wedge Ret = 2 * A * Math.pow(2, N) * N) \vee$$
$$(N > 0 \wedge A \neq 0 \wedge N = 1 \wedge Ret = (1 + Math.pow(2, 400 * A))/(200 + (400 * A))) \vee$$
$$(N > 0 \wedge A \neq 0 \wedge N = 2) \wedge$$
$$Ret = ((1 + Math.pow(2, 400 * A)) + Math.pow(2, 800 * A))/(200 + (800 * A)))$$

Versions. We denote by m and m' two successive versions of a method. We assume that m and m' have the same method name and input parameters (otherwise – they are not equivalent). We consider *common* all statements that are syntactically identical in m and m'. Statements added in m' are referred to as *insertions* and statements removed for m are referred to as *deletions*; we represent statement updates as a deletion of an old statement and an insertion of a new one. For the example in Figure 1, statements in lines 11 and 17 are updates, represented by deletion and insertion of the corresponding statements in m and m'.

Symbolic-Execution-Based Equivalence Checking. Two input methods m and m', with symbolic summaries M and M', respectively, are *functionally equivalent* if M is logically equivalent to M'. An *equivalence assertion* is a first-order logic formula Φ that helps determine such equivalence [38]: $\Phi = \neg(M \Leftrightarrow M')$.

This formula is typically given to a SAT or SMT solver [14, 18], which either proves that no satisfying assignment to this formula exists, meaning that m and m' are equivalent, or finds a counterexample to demonstrate non-equivalence. That is, the *satisfiability* of Φ indicates that m and m' produce different outputs for at least one input. If a solver cannot find any satisfying assignment for Φ – i.e., the result is UNSAT – the methods are equivalent.

Symbolic summaries can contain *uninterpreted functions*, i.e., functions that are free to take any value [29]. The equality logic with uninterpreted functions relies on functional consistency – a conservative approach to judging functional equivalence which assumes that instances of the same function return the same value if given equal arguments. We leverage this quality of uninterpreted functions to abstract portions of common code and also to model method calls.

Symbolic-execution-based equivalence checking approaches rely on SAT or SMT solvers, such as Z3 [17], to find satisfying assignments for equivalence assertions. Yet, as the satisfiability problem with non-linear constraints is generally undecidable and practically difficult, our goal is to simplify these formulas and eliminate a large portion of 'unknown' results.

3 MOTIVATING EXAMPLE

In this section, we use the example in Figure 1 to describe two existing solutions for method-level functional equivalence, DSE and IMP-S, and outline their limitations. We introduce our solution that addresses these limitations in the following section.

Differential symbolic execution (DSE). Person et al. [38] are among the first to use symbolic execution for program equivalence checking. DSE uses *uninterpreted functions* to abstract common parts of the compared code, thus skipping portions of the program that are identical in two versions and reducing the scope of the analysis. For the example in Figure 1, there are three common code blocks: in lines 2-5, line 7, and lines 13-16. The return statements (lines 8 and 21 in both versions), even if common, are not abstracted as they capture the effect of the entire path and are required by the symbolic execution engine for producing the summary.

For each common block, the tool collects all variables that are *defined* in the block and represents each variable as an uninterpreted function which accepts as inputs all variables that are *used* in the block. For example, for the block in line 7, *res* is the output which is represented by an uninterpreted function: $UF^{7}_{res}(A, N)$.

The benefits of using uninterpreted functions can be realized when two symbolic summaries are compared with each other: in this example, the equivalence assertion for establishing equivalence of these two paths reduces to $\neg(UF^{7}_{res}(A, N) \Leftrightarrow UF^{7}_{res}(A, N))$, which can be determined unsatisfiable using the theory of uninterpreted functions [29]. As in evolving software common code blocks are expected to appear more frequently than changed code, such an approach has a potential to "hide" loops and complex expressions, leading to more "solved" equivalence cases and more "complete" solutions than that of a "naïve" checker with loop bounding.

However, as discussed in Section 1, abstracting all syntactically identical code is not effective for two reasons. First, even if a solver determines that an equivalence assertion is satisfied, i.e., the method summaries are non-equivalent, this can be a false-negative result if the satisfying assignment allocates to an uninterpreted function a value that it cannot take in practice. For example, the uninterpreted function $UF^{2}_{acc}()$ representing the code in line 2 of Figure 1 cannot take any value other than 200. Moreover, introducing numerous uninterpreted functions may result in unnecessary complexity in symbolic summaries, producing 'unknown' results that could be eliminated if the abstracted code is simple, like in lines 2 and 3 of Figure 1. Thus, there is a need for a decision process establishing which *parts* of the common code need to be abstracted away and which are not. We address this need in our work.

IMPacted Summaries (IMP-S) [6]. Instead of identifying common code blocks, Bakes et al. [6] propose a technique that uses static analysis, namely, forward and backward control- and data-flow analysis, to identify all statements impacted by the changed code. The tool then prunes all clauses of symbolic summaries that do not

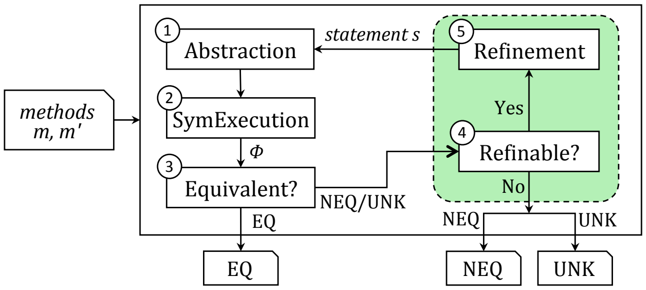

Figure 2: ARDiff Architecture.

contain any impacted statements. For the example in Figure 1, statements in lines 4, 5, 11, 15, 17, and 19 are impacted by the change in line 11. Since only the statements in lines 6, 7, and 8 are used in the summary of the path in lines 2-9 ($N \leq 0 \wedge$ *Ret=A*Math.pow(2,N)*) and none of these statements are impacted by the change, the summary of this path can be pruned from the method's symbolic summary altogether.

The main limitation of this approach lies in the assumption that all impacted statements are required for deciding equivalence of two programs. This assumption results in inclusion of unnecessary statements and constraints in the symbolic summary. For example, the path in lines 2-5, 12-19 contains the impacted statement in line 15. As such, IMP-S keeps the complex formula introduced by this statement, *Math.pow*, in the symbolic summary and, as a result, also has to bound the *for* loop that controls this statement (lines 13-16). Due to the complexity introduced by the statement, the output of the tool for this case is 'unknown'. Yet, the statement in line 15 is common between the two versions of the program and can, in fact, be abstracted away, without hindering the decision about the equivalence of the methods. That is, the approach (a) leads to unnecessary 'unknowns' and also (b) often requires bounding loops, even when the execution of the loop can be abstracted away altogether. Like in the case of DSE, establishing which parts of the statements are required for determining equivalence, without inflating symbolic summaries, is a challenging task.

4 OUR APPROACH

In this section, we provide a high-level overview of ARDiff and demonstrate its operation on the methods m and m' in Figure 1. We then describe its main process – selecting refinement candidates – in detail. Finally, we formally prove the correctness of the output produced by the tool.

4.1 ARDiff Overview

ARDiff obtains as input two versions of a method, m and m', and reports whether these versions are equivalent (denoted by EQ) or non-equivalent (denoted by NEQ). If equivalence cannot be established, it returns unknown (denoted by UNK).

A high-level overview of ARDiff is given in Figure 2. It is inspired by the CEGAR abstraction/refinement loop [13], aiming to arrive at the optimal abstraction which hides complex statements while refining statements needed to prove equivalence. As the first step, ARDiff abstracts all syntactically equivalent statements in m and m' using uninterpreted functions, as done in DSE [38] and discussed in Section 3. It then produces symbolic summaries for the abstracted methods (denoted by M and M') and generates the equivalence checking assertion $\Phi = \neg(M \Leftrightarrow M')$ (step 2 in Figure 2).

For the example in Figure 1, ARDiff abstracts three common blocks: in lines 2-5, 7, and 13-16. That produces seven uninterpreted functions: $UF^{2}_{acc}()$, $UF^{3}_{res}()$, $UF^{4}_{bess}(norm)$, $UF^{5}_{twoarg}(arg)$, $UF^{7}_{res}(arg, norm)$, $UF^{14}_{bess}(arg, norm)$, $UF^{14}_{bess}(acc, twoarg, norm)$, and $UF^{15}_{res}(res, bess, norm)$. The input parameters of these uninterpreted functions will be replaced by their corresponding symbolic values during the symbolic execution; the produced assertion Φ, with three symbolic paths in both M and M', is shown in Figure 3a.

Next, Φ is passed to an SMT solver (step 3 in Figure 2). If the solver determines that it does not have a satisfying assignment, i.e., the functional summaries of the input methods with uninterpreted functions are equivalent, ARDiff outputs EQ and the process terminates. In this case, the soundness of the abstraction guarantees that the concrete methods are also equivalent (see Section 4.3).

Otherwise, the result is either NEQ or UNK. If Φ contains uninterpreted functions, the UNK result might be an artifact of abstraction and using a subset of original statements with concrete values might result in simpler summaries. For the NEQ case, a satisfying assignment making the summaries non-equivalent might assign values to uninterpreted functions even though code abstracted by these functions can never produce such values [27]. For the example in Figure 3a, Φ is satisfiable (NEQ result), even though methods m and m' are, in fact, equivalent. Such results occurs because assigning a value other than 200 to $UF^{2}_{acc}()$ will make the formulas M and M' different.

In most cases, refining abstractions that lead to NEQ or UNK results can help eliminate false-negatives and unresolved instances. ARDiff then checks whether refining Φ is effective (step 4 in Figure 2). For the UNK case, this simply translates into checking whether Φ still contains uninterpreted functions. For the NEQ case, the tool checks whether $\neg\Phi$ is satisfiable, that is, whether there exists at least one assignment that makes M and M' equivalent. If so, ARDiff proceeds to the refinement step. Otherwise, further refinement is either impossible or ineffective; the tool then returns the corresponding result to the user and the process terminates.

This refinement step (step 5 in Figure 2) is at the core of our approach: it accepts as input the formula Φ and, by applying a set of heuristics, outputs a statement s in methods m and m' that is skipped from being abstracted away. ARDiff then proceeds to creating a finer-grained abstraction (next iteration of step 1 in Figure 2), aiming at producing symbolic summaries where only the code that is required to establish equivalence appears in the summaries in its refined form.

4.2 The Refinement Process

To identify the best refinement candidate in each iteration, we utilize a set of heuristics, described in Algorithm 1. The main goal of these heuristics is to find the most "critical" yet simple code statements that can help establishing equivalence/non-equivalence without introducing unnecessary complexity into the equivalence assertion. Our heuristics work on two levels: symbolic summaries (Heuristic 1 and 2) and the code itself (Heuristic 3). The goal of summary-level Heuristics 1 and 2 is to narrow down the selection

$$
\begin{aligned}
&\big((N \le 0 \land Ret = UF^{7}_{res}(N,A)) \ \lor\ (N > 0 \land A = 0 \land Ret = UF^{5}_{twoarg}(A) * UF^{4}_{bess}(N)) \lor \\
&(N > 0 \land A \neq 0 \land Ret = UF^{15}_{res}(UF^{3}_{res}(), UF^{14}_{bess}(UF^{2}_{acc}(), UF^{5}_{twoarg}(A), N), N)/(\boldsymbol{200} + UF^{14}_{bess}(UF^{2}_{acc}(), UF^{5}_{twoarg}(A), N)))\big) \qquad \Leftrightarrow \\
&\big((N \le 0 \land Ret = UF^{7}_{res}(N,A)) \ \lor\ (N > 0 \land A = 0 \land Ret = UF^{5}_{twoarg}(A)) \lor \\
&(N > 0 \land A \neq 0 \land Ret = UF^{15}_{res}(UF^{3}_{res}(), UF^{14}_{bess}(UF^{2}_{acc}(), UF^{5}_{twoarg}(A), N), N)/(\boldsymbol{UF^{2}_{acc}()} + UF^{14}_{bess}(UF^{2}_{acc}(), UF^{5}_{twoarg}(A), N)))\big)
\end{aligned}
$$

(a) Refinement Iteration #1.

$$
\begin{aligned}
&\big((N \le 0 \land Ret = UF^{7}_{res}(N,A)) \ \lor\ (N > 0 \land A = 0 \land Ret = UF^{5}_{twoarg}(A) * \boldsymbol{UF^{4}_{bess}(N)}) \lor \\
&(N > 0 \land A \neq 0 \land Ret = UF^{15}_{res}(UF^{3}_{res}(), UF^{14}_{bess}(200, UF^{5}_{twoarg}(A), N), N)/(200 + UF^{14}_{bess}(200, UF^{5}_{twoarg}(A), N)))\big) \qquad \Leftrightarrow \\
&\big((N \le 0 \land Ret = UF^{7}_{res}(N,A)) \ \lor\ (N > 0 \land A = 0 \land Ret = UF^{5}_{twoarg}(A)) \lor \\
&(N > 0 \land A \neq 0 \land Ret = UF^{15}_{res}(UF^{3}_{res}(), UF^{14}_{bess}(200, UF^{5}_{twoarg}(A), N), N)/(200 + UF^{14}_{bess}(200, UF^{5}_{twoarg}(A), N)))\big)
\end{aligned}
$$

(b) Refinement Iteration #2.

Figure 3: The Equivalence Assertions $\Phi = \neg(M \Leftrightarrow M')$ for Methods m and m' in Figure 1.

Algorithm 1: The Refine Procedure

```
1  Input   : Equivalence assertion $\Phi = \neg(M \Leftrightarrow M')$
   Output : statement $s$ to skip
2  begin
3      $U \leftarrow Un(\Phi)$;                    ▹ Consider all uninterpreted functions in $\Phi$
4      $U_c \leftarrow \emptyset$                   ▹ Refinement candidates
5      foreach $u \in U$ do
           ▹ Heuristic 1: Is there a value of $u$ that make the summaries equivalent
             for any values of the remaining functions?
6          if $SMT[\exists u | \forall_{u_i \in U \setminus \{u\}}, M \Leftrightarrow M'] =$ 'SAT' then
7              $U_c \leftarrow U_c \cup \{u\}$;          ▹ Add $u$ to the set of candidates
           ▹ Heuristic 2: Is $u$ used differently by $M$ and $M'$?
8          if $Count(u,M) \neq Count(u,M')$ then
9              $U_c \leftarrow U_c \cup \{u\}$;          ▹ Add $u$ to the set of candidates
10     if $U_c = \emptyset$ then
11         $U_c \leftarrow U$;        ▹ Cannot narrow down selection; consider all functions
       ▹ Heuristic 3: Rank all statements (lower is better)
12     $S \leftarrow \bigcup_{u \in U_c}$ Statements($u$);     ▹ All statements from all functions in $U_c$
13     $R \leftarrow \emptyset$;                         ▹ Ranked statements
14     foreach $s \in S$ do
           ▹ Heuristic 3.1: The depth of $s$ in loop nesting
15         $r_1 \leftarrow$ LoopNestingIndex($s$);
           ▹ Heuristic 3.2: The total number of non-linear arithmetic operators in $s$
16         $r_2 \leftarrow$ NonLinearOperators($s$);
17         $r \leftarrow r_1 + r_2$;                     ▹ Total rank
18         $R \leftarrow R \cup \{[s, r]\}$;        ▹ Add $s$ with rank $r$ to candidate set
19     return smallestRank(R);        ▹ Return a statement with the smallest rank
```

only to statements truly necessary to prove/disprove equivalence; these heuristics do not directly reason about the structure of the code. Then, code-aware Heuristic 3 selects the next refinement candidate based on code simplicity.

We start the description from the summary-level heuristics, which consider all uninterpreted functions in Φ as potential refinement candidates (line 3) and further rank them to identify the best refinement candidates U_c (lines 4-11):

Heuristic 1: First, for each uninterpreted function $u \in U$, we check whether there exists a value of u that would make the formula Φ hold, regardless of the values of other functions. The rationale behind this heuristic is that if such value exists and refinement will prove that it can hold, no other functions need to be refined, eliminating the need to introduce unnecessary complexity. For example, given three uninterpreted functions u_1, u_2, and u_3, and $\Phi = \neg((u_1 * u_2 * u_3) \Leftrightarrow ((u_1 + u_2) * u_3))$, setting u_3 to 0 can make these two summaries equivalent. Refining this function can help to prove the equivalence of the summaries without further refining u_1 and u_2.

To check if such a value exists, for each $u \in U$, we build the formula $[\exists u \cdot \forall_{u_i \in U \setminus \{u\}} \cdot M \Leftrightarrow M']$ and check if it is satisfiable by passing it to an SMT solver. If the answer is yes, we add the function to the list of candidates U_c (lines 6-7).

Heuristic 2: Our second summary-level heuristic is based on the intuition that functions that are used differently in M and M' are better candidates for refinement because they are more likely to lead to nonequivalent summaries. In that case, again, non-equivalence can be established without refining the remaining functions. For example, given uninterpreted functions u_1 and u_2, and $\Phi = \neg((u_1 + u_2) \Leftrightarrow (5 + u_2))$, satisfaction of the formula can be established by refining u_1. If u_1 is equal to 5, the summaries are equivalent regardless of the value of u_2.

To follow on this intuition, for each $u \in U$, we count the number of occurrences in M and M'. If the number differs, we add the function to the list of candidates U_c (lines 8-9). For the example in Figure 3a, this heuristic will identify functions $UF^{2}_{acc}()$ and $UF^{4}_{bess}(norm)$. The first is selected because acc is used in line 17 of m' in Figure 1 but not in m. The second is selected because $bess$ is used in line 11 of m in Figure 1 but not in m'.

When no heuristics identifies promising refinement candidates to add to U_c, we set U_c to all uninterpreted functions in U (lines 10-11 in Algorithm 1). We then proceed to the next step: analyzing code-level information for identifying the most promising statement candidate to skip (lines 12-19).

Heuristic 3: To perform code-level analysis, we extract all statements abstracted by the uninterpreted functions in U_c (line 12). In our example, these are statements in lines 2 and 4 in Figure 1. Then, for each statement, we calculate two metrics. The first (Heuristic 3.1) returns the depth of the statement in the nested loop structure (line 15). The rationale behind this metric is that statements that are not nested in any loops are better candidates for refinement.

For the example in Figure 1, both statements in lines 2 and 4 have a nesting index of 0 – they are not nested inside any loop. In fact, in this example, only statements in lines 14 and 15 have a nesting index of 1.

Next (Heuristic 3.2), we calculate the number of non-linear arithmetic operators, such as multiplication, division, power, and square root, in a statement (line 16). The rationale is, again, that simpler statements which do not introduce additional complexity for an SMT solver are better candidates for refinement [16]. For our example in Figure 1, the statement in line 2 has no such operators and the statement in line 4 has 2: *pow* and *.

We sum the loop nesting index and the statement complexity index and consider that to be the score of a statement (lines 17-18 in Algorithm 1). After all statements are scored, this process returns a statement with the smallest score (line 19), choosing one at random if multiple statements with the same score exist. In our example, the statement in line 2 of Figure 1 has a score of 0 and is returned by the procedure; it will not be abstracted in the next iteration.

The equivalence checking assertion produced after this refinement is shown in Figure 3b. When given to an SMT solver (ARDiff's step 2 in Figure 2), the result is still NEQ. As the formula still contains uninterpreted functions, ARDiff proceeds to the second refinement iteration. In this case, Φ contains 6 uninterpreted functions: all listed above besides $UF^{2}_{acc}()$.

If $UF^{5}_{twoarg}(arg)$ is assigned a value of 0, Φ is unsatisfiable regardless of the value of other functions. Thus, it is picked by Heuristic 1 and added to U_c. For Heuristics 2, $UF^{4}_{bess}(norm)$ is selected again, like in the previous iteration.

The statements abstracted by these uninterpreted functions are the statements in lines 4 and 5 of Figure 1. The rank of the first one is 2 and of the second is 1. Thus, Heuristic 3 will pick the statement in line 5 as the next candidate. To prove equivalence, we need to show that the value of *res* in line 11 is the same in both methods. As predicted by Heuristic 1, that is indeed the case, because the skipped statement in line 5 shows that *twoarg = 2* A* and, thus, under the path condition of $N > 0 \land A = 0$, *res* is 0 in both cases.

After this refinement, the process terminates as the equivalence of the methods is established, without refining the remaining uninterpreted functions. The order of refinement plays a key role here, as refining the function $UF^{4}_{bess}(norm)$ first would lead to an 'unknown' result. Thus, our heuristics were effective in choosing the right refinement candidates.

We evaluate each of the proposed heuristics separately, as well as their combination, comparing our results to that of existing tools, in Section 5. Next, we show that the results produced by ARDiff are provably correct.

4.3 Validity of the Results

We denote by M_f and M'_f the symbolic summaries of methods m and m', respectively, that contain no uninterpreted functions. We denote by M_u and M'_u symbolic summaries of these methods that might contain uninterpreted functions. When ARDiff terminates with an EQ result and the equivalence assertion still contains an uninterpreted function, $\Phi = \neg(M_u \Leftrightarrow M'_u)$ is UNSAT. This means that $M_u \Leftrightarrow M'_u$ is *valid*, i.e., satisfied by every assignment. According to Kroening and Strichman [29], uninterpreted functions only "weaken" the formula; thus $M_f \Leftrightarrow M'_f$ is also valid and the methods are equivalent. ARDiff terminates with a NEQ result and uninterpreted functions in the equivalence assertion only if $M_u \Leftrightarrow M'_u$ is UNSAT. In that case, $\neg(M_u \Leftrightarrow M'_u)$ is valid, which implies that $\neg(M_f \Leftrightarrow M'_f)$ is valid. That is, there exists no assignment making M_f and M'_f equivalent, i.e., the original methods are not equivalent.

4.4 Implementation

To identify common vs. changed code blocks, we use GumTree [20] – a state-of-the-art code differencing tool for languages such as Java, C, and Python. GumTree identifies inserted, deleted, changed, and moved code statements, using an Abstract Syntax Tree (AST) structure [46] rather than a text structure. We consider all the remaining statements *common* and, in each refinement iteration, exclude from this set statements that our algorithm chose to refine.

We group the remaining statements into consecutive blocks and, for each block, identify subsets of statements that can be abstracted by uninterpreted functions. As discussed in Section 2, some common statements cannot be abstracted away, e.g., return statements or conditionals that control return statements and changed blocks. For example, common statements in lines 2-10 in Figure 1 corresponds to two "abstractable" common blocks: in lines 2-5 and line 7. We then use ASM-DefUse [3] – an extension to the ASM analysis framework [7], to identify inputs and outputs of common blocks and abstract each variable defined in the block with an uninterpreted function.

We use the Java PathFinder symbolic execution framework (JPF-SE) [37] and the Z3 SMT solver [17] for producing and reasoning about symbolic summaries. We configure Z3 to use *simplify* and *aig* tactics for compressing Boolean formulas, and *qfnra-nlsat* and *smt* tactics for handling non-linear arithmetic. An up-to-date, fully-functional implementation of ARDiff is available in our online appendix [5]; the latest citable release can also be found online [4].

5 EVALUATION

In this section, we discuss our experimental setup and evaluation results. Our goal is to answer the following research questions:

RQ1. How effective are the heuristics applied by ARDiff?
RQ2. How does the effectiveness of ARDiff compare with that of existing solutions?

In what follows, we describe our experimental subjects, methodology, and findings. We then discuss threats to the validity of our results. To facilitate reproducibility, our experimental package is available online [5].

5.1 Subjects

We started by using benchmarks proposed by recent work on symbolic-execution-based equivalence checking [34, 45], which we refer to as the ModDiff benchmarks. As these benchmarks are relatively small (28 cases, 7.4 statements per case on average) and contain no complex constraints, we also adapted for our evaluation benchmarks collected by Li et al. [32] from the literature on evaluating symbolic and concolic execution methods in the presence of complex path conditions and non-linear functions [19, 44]. The methods of those benchmarks are classical numerical computation

Table 1: Evaluation Benchmarks.

Bench.	# M	LOC			% Non-Linear	# Loops	% Changed
		Min.	Max.	Mean	Exp.		Stms
ModDiff	28	4	14	7.4	0	0.9	26
airy	2	5	13	9	0	0	40
bess	17	4	60	21.4	46.4	0.5	10
caldat	2	22	45	33.5	27.4	1.5	9
dart	1	9			9.1	0	22
ell	10	6	79	37.9	34.1	1.5	8
gam	9	7	51	22.5	32.1	1.2	12
pow	1	22			4.8	0	4
ran	8	7	87	34.4	40.7	2.5	5
sine	1	148			7.8	0	0.2
tcas	3	11	19	13.6	0	0	19
tsafe	3	9	32	22.6	33.3	0	6
Total	85	4	148	33.9	31.4	0.89	12

functions used in real-world distributions. We excluded from this suite methods with less than 3 lines of code, as we cannot effectively inject changes in these methods.

We further excluded 14 methods that contain string and array manipulations: even though the relevant decision procedures (ABC and z3str) are integrated with JPF-SE, its support for strings and arrays is still incomplete. We thus cannot provide full support for these constructs at the moment. Yet, supporting strings, arrays, and other language constructs is orthogonal to the abstraction/refinement idea proposed in this work. In fact, Heuristics 1, 2, and 3.1 (loops) are code-agnostic and will work with any code constructs. Heuristic 3.2 currently only considers arithmetic operations but can easily be extended, e.g., to count access to strings/arrays with a symbolic index as another source of complexity.

The remaining 28 ModDiff benchmarks and 57 benchmarks from the work of Li et al. are listed in Table 1. The first three columns of the table show the name of each benchmark, the number of methods it includes, and the number of lines of code (LOC) in benchmark's methods – minimum, maximum, and mean. The fourth and fifth columns of the table show the fraction of statements with complex non-linear arithmetic and the number of loops in each method, averaged across all methods of a particular benchmark. For example, the *bess* benchmark used as the baseline for our motivating example in Figure 1 contains 17 methods ranging from 4 to 60 LOC, with 46.4% of complex statements on average. Overall, considering methods from all benchmarks together, 58% of the methods (50 out of 85) contain at least one loop and there are 0.89 loops per method on average. 52.9% of the methods (45 out of 85) contain at least one statement with complex non-linear arithmetic and there is 31.4% of statements with non-linear arithmetic per method on average.

As these benchmarks were not designed for the equivalence checking problem, we had to create an equivalent and non-equivalent version of each method. To this end, we systematically injected changes to each method, using the following protocol: first, we used a random number generator software to automatically pick the number of changes to inject in each method: between one and three. Then, we used the software to select the location for the change. To produce *non-equivalent* cases, we relied on a catalog of changes proposed by Ma and Offutt [33] and picked a type of change (insertion, deletion, update) and the essence of a change (arithmetic operation modification or condition modification).

For *equivalent cases*, we first attempted to use one of the existing code refactoring techniques: split loops, extract variable, inline variable, consolidate conditional fragments, decompose conditionals, or replace nested conditional with guard clauses [2, 24]. If none of these modifications were applicable, we inserted dead code, such as redundant assignments or unreachable code guarded by conditions that cannot hold in practice. We balanced the number of complex and simple non-linear logic expressions in all statements we generated. The last column of Table 1 shows the fraction of changed statements in each of the benchmarks, averaged across all methods of a benchmark.

The benchmarks contributed in this work are substantially larger and more comprehensive than those used in prior studies on equivalence checking techniques. Our benchmarks, together with a detailed description of the injected changes for each benchmark are available online [5].

5.2 Methods and Metrics

To answer **RQ1**, we created three versions of our tool, which differ by the heuristics they apply in the Refinement step (step 5 in Figure 2):

(1) ARDiff$_R$ selects a statement to preserve at random, skipping all the heuristics described in Section 4.2.
(2) ARDiff$_{H_3}$ applies only the statement-level heuristic (Heuristic 3), picking the "simplest" statement to preserve while considering all uninterpreted functions as refinement candidates.
(3) ARDiff applies summary-level heuristics (Heuristic 1 and 2) to narrow down the set of candidate uninterpreted functions to refine and then applies the statement-level heuristic for picking the statement to refine (Heuristic 3).

For each of the tool versions, we counted the number of cases where the tool could correctly prove or disprove equivalence, for equivalent and non-equivalent cases separately. We also counted the number of iterations it took the tool for producing the right answer. Finally, we recorded the execution time for each of the benchmarks.

To answer **RQ2**, we compared the version of the tool that performed the best in RQ1 to two state-of-the-art method-level equivalence checking techniques: DSE and IMP-S. We excluded from our evaluation regression-verification-based tools, such as SymDiff [31] and RVT [26], as these tools can only prove equivalence but cannot disprove it. We could not compare our technique with Rêve [22] as this tool cannot handle programs containing doubles, which is the majority of our benchmarks. Like for RQ1, we counted the number of correctly solved cases and the execution time of each tool.

We reached out to the authors of DSE and IMP-S, but the implementations of the techniques were not available at the time of writing. We thus re-implemented the techniques and included them in our generic equivalence checking framework [5]. We ran the re-implemented techniques on all examples given in the corresponding papers to ensure correctness. Furthermore, two authors manually cross-validated the results of all experiments on all tools.

We used the same setup to configure all tools, setting a timeout of 300 seconds for each tool, which included a timeout of 100 seconds for the Z3 assertion checking step. We ran all our experiments on an Ubuntu 18.04.4 Virtual Machine (VM), with 4 cores and 16 GB of

Table 2: Correctly Resolved Cases, With and Without Bounding.

Bench	#M	Equivalent					Not Equivalent				
		DSE	IMP-S	ARDiff$_R$	ARDiff$_{H_3}$	ARDiff	DSE	IMP-S	ARDiff$_R$	ARDiff$_{H_3}$	ARDiff
ModDiff	16	14	16	16	16	16	-	-	-	-	-
	12	-	-	-	-	-	12	12	12	12	12
airy	2	2	2	2	2	2	2	2	2	2	2
bess	17	6	10+3	10+1	13+1	15+1	4	6	6	7	7
caldat	2	1	1	2	2	2	0	0	1	1	1
dart	1	1	1	1	1	1	1	1	1	1	1
ell	10	2	1	2+1	3	3+1	1	2	1	3	3
gam	9	3	4+2	4	4+1	7	3	3	4	4	4
pow	1	1	1	1	1	1	0	1	1	1	1
ran	8	3	2+3	5+1	7+1	7+1	2	5	3	4	4
sine	1	0	0	0	0	0	0	0	0	0	0
tcas	3	2	3	2	3	3	2	3	2	2	2
tsafe	3	0	2	2	2	3	0	1	1	1	1
Total	73	35	43+8	47+3	54+3	**60+3**	-	-	-	-	-
	69	-	-	-	-	-	27	36	34	**38**	**38**

Table 3: Mean Runtime in Seconds.

Bench	#M	Equivalent					Not Equivalent				
		DSE	IMP-S	ARDiff$_R$	ARDiff$_{H_3}$	ARDiff	DSE	IMP-S	ARDiff$_R$	ARDiff$_{H_3}$	ARDiff
ModDiff	16	6.46	7.23	7.20	7.16	7.17	-	-	-	-	-
	12	-	-	-	-	-	8.29	8.31	8.44	8.39	8.42
airy	2	2.21	2.68	2.42	2.40	2.39	3.13	3.87	3.19	3.2	3.07
bess	17	20.55	42.54	113.46	60.27	25.04	75.51	77.22	182.81	165.52	148.25
caldat	2	17.37	15.46	34.28	29.28	18.79	150.90	152.09	154.86	154.24	153.74
dart	1	2.69	2.72	2.71	2.69	2.72	1.9	2.75	2.61	2.73	2.81
ell	10	139.58	251.98	223.02	234.82	194.59	155.99	169.91	270.76	230.50	209.57
gam	9	22.84	134.26	148.66	116.32	78.18	81.43	130.35	205.34	205.39	205.13
pow	1	1.92	2.34	1.98	1.99	1.96	2.01	3.07	12.93	11.83	12.97
ran	8	1.7	29.34	78.77	8.95	7.68	151.83	53.07	202.81	168.83	168.68
sine	1	300	300	300	300	300	300	300	300	300	300
tcas	3	2.95	3.92	101.08	5.19	3.86	13.93	28.81	101.89	101.26	102.1
tsafe	3	2.19	104.43	15.89	15.31	11.59	102.34	35.66	205.78	203.26	203.26
Mean	-	**31.05**	78.79	95.34	67.49	49.93	-	-	-	-	-
	-	-	-	-	-	-	80.94	**79.99**	158.4	144.33	132.86

RAM, which was hosted on an Ubuntu 16.04 server with 64 cores and 512GB of memory. We enforced the timeout of 300 seconds on each process by using the Linux 'timeout' command and used 'user time' as reported by the Linux 'time' command to measure time for terminating processes. We used the Java –Xms option to control memory allocation.

5.3 Results

Table 2 shows the number of correctly resolved cases for equivalent and non-equivalent methods of each benchmark, separately. For equivalent cases, we also distinguish between cases that were resolved without loop bounding and the cases where loop bounding was required. We report our result per benchmark and also in total for all benchmarks. For example, for the *bess* benchmark in line 4 of Table 2, DSE is able to correctly resolve six out of 17 equivalent cases; IMP-S is able to correctly resolve 10 cases without loop bounding and three more cases with loop bounding. ARDiff$_R$, ARDiff$_{H3}$, and ARDiff resolve 10, 13, and 15 cases, respectively, without loop bounding and one more case each with loop bounding. For non-equivalent cases, DSE and IMP-S are able to disprove equivalence in four and six out of 17 cases, respectively; the three variants of our tool are able to resolve six, seven, and seven cases.

Table 3 shows the mean execution time, in seconds, averaged over all cases of each benchmark, for equivalent and non-equivalent cases separately.

RQ1. Comparing the performance of the three versions of our tool with each other shows that the combination of all heuristics that ARDiff applies is the most beneficial for resolving both equivalent and non-equivalent cases: ARDiff is able to resolve 63 equivalent cases, compared with 57 for ARDiff$_{H3}$ and only 50 for ARDiff$_R$. This includes three bounded cases for each tool. Interestingly, for the *gam* benchmark, while ARDiff$_{H3}$ had to bound one case, ARDiff could avoid selecting the statement s leading to the loop bounding. That is because it applied Heuristic 2 first, to select an uninterpreted function that only appeared in the summary of the changed method m'. Even though statements of this function had a higher individual complexity score than s, skipping them in the abstraction helped prove equivalence without any need to refine loops, as predicted by the heuristics. ARDiff was able to resolve two additional cases for this benchmark, both without loop bounding.

Table 4 shows the total number of cases where performing additional refinement iterations helped each of the tool version arrive at

Table 4: Correct Cases with ≥ 1 Iterations.

#M	ARDiff$_R$		ARDiff$_{H_3}$		ARDiff	
	# Cases	# Iter.	# Cases	# Iter.	# Cases	# Iter.
EQ (73)	15	4.2	22	2.7	28	1.6
NEQ (69)	6	4.4	10	4.2	10	3.3

the correct result (# Cases) as well as the mean number of iterations per case (# Iter). The number of uninterpreted functions in the final summary for each case and the time spent in the refinement step are also available in our online appendix [5].

While ARDiff$_R$ has the highest mean number of iterations for both equivalent and non-equivalent cases, it is able to solve the smallest number of cases. That is because by making a "wrong" pick, it arrives at a solution that leads to an 'unknown' result and keeps refining the summary until it times out. ARDiff$_{H3}$ makes "smarter" choices and is thus able to solve more cases with a lower number of iterations. The combination of heuristics applied by ARDiff allows it to reach the best result with the smallest number of iterations. As a result, ARDiff also outperforms other variants in the mean execution time, as shown in Table 3.

Answer to RQ1: To summarize, our experiments show that the refinement heuristics implemented by ARDiff increase its effectiveness, in terms of the total number of equivalent and non-equivalent cases the tool can resolve, and its efficiency, in terms of both the execution time and the number of iterations per benchmark.

RQ2. We now compare the performance of ARDiff to that of DSE and IMP-S. Naturally, ArDiff outperforms DSE because it extends DSE with the abstraction-refinement loop and the refinement heuristics. As a result, ARDiff solves 28 more equivalent and 10 more non-equivalent cases compared with DSE. However, it is also slower than DSE because it has to perform more abstraction-refinement iterations to achieve these results.

When comparing with IMP-S, ARDiff can solve 63 out of 73 (86%) equivalent cases vs. 51 cases (69%) for IMP-S. It also had to bound loop iteration in only 3 vs. 8 cases. All equivalent cases solved by IMP-S are solved by ARDiff as well. In addition, there are 12 cases solved by ARDiff and not by IMP-S. That is because IMP-S relies on static analysis, which over-approximates the set of statements really required to prove/disprove equivalence. As such, it deems a complex statement impacted and includes it in the summary, even though the statement is unnecessary for proving equivalence, as we showed in Section 3 for our motivating example.

Non-equivalent cases are a harder challenge for any equivalence checking tool; ARDiff performs comparably and even slightly better than IMP-S, solving 38 vs. 36 non-equivalent cases. In five cases, from *bess*, *caldat*, *gam*, and *ell* benchmarks, ARDiff could produce the correct proof when IMP-S resulted in 'unknowns', due to the overapproximations described above. However, there are three non-equivalent cases solved by IMP-S, which result in 'unknown' for ARDiff: in *tcas*, *bess*, and *ran* benchmarks. In all three cases, IMP-S was more successful because the change only impacted a very small portion of each method. As such, IMP-S could quickly prove non-equivalence while ARDiff continued refining numerous uninterpreted functions in these methods until it ran out of time. Interestingly, increasing the execution time allowed ARDiff to solve the previously unresolved case in the *tcas* benchmark as well. For the other two cases, even though the solver made a correct non-equivalence decision in one of the iterations, ARDiff conservatively refined uninterpreted functions (see Figure 2) until it obtained an equivalence assertion which is no longer solvable.

For the runtime, ARDiff outperforms IMP-S in terms of the execution time for equivalent cases: 49.93 vs. 78.79 seconds per case, on average. That is because it is able to successfully solve more cases, eliminating many timeouts. For non-equivalent cases, IMP-S's performance is higher. The main portion of performance loss in our tool occurs in 'unknown' cases: while IMP-S makes one attempt and terminate if the SMT solver produces 'unknown', ARDiff will attempt to refine the assertion and try multiple times. Yet, this design choice allows ARDiff to solve more cases. Interestingly, for cases solved by both ARDiff and IMP-S, the performance of the tools is comparable – 14.77s and 14.56s on average, respectively.

Answer to RQ2: To summarize, ARDiff substantially outperforms existing techniques for equivalent cases and performs comparably, and even slightly better, for non-equivalent cases. The increased accuracy comes at the cost of a decrease in execution time when compared with DSE and with IMP-S for non-equivalent cases only. ARDiff outperforms IMP-S in terms of execution time for equivalent cases.

5.4 Threats to Validity

For **external validity**, our results may be affected by the selection of subject methods that we used and may not necessarily generalize beyond our subjects. We attempted to mitigate this threat by using a set of benchmark methods available from related work on symbolic-execution-based equivalence checking and by extending this set to include additional benchmarks for evaluating symbolic execution methods in the presence of complex conditions, such as loops and non-linear arithmetic functions. As we used a set of different benchmarks of considerable size and complexity, we believe our results are reliable.

As we had to inject changes when generating equivalent and non-equivalent versions for each of these new benchmarks, the changes may not reflect real cases of software evolution. We mitigated this threat by basing our changes on existing refactoring techniques. We mitigated possible investigator bias of creating these cases by applying the changes in a systematic way that considered a broad range of change types and a random number generator software to pick the change type and location.

Finally, we had to re-implement the baseline tools, DSE and IMP-S. To ensure the correctness of our implementation, we run the re-implemented techniques on all examples given in the corresponding papers. Furthermore, as the implementation of ARDiff relies on the same underlying framework and setup, we do not believe that hinders the validity of our findings.

For **internal validity**, deficiencies of the underlying tools our approach uses, such as the symbolic execution engine and SMT solver, might affect the accuracy of the results. We controlled for this threat by manually analyzing the cases that we considered and confirming their correctness.

6 DISCUSSION AND RELATED WORK

Our discussion of related work focuses on techniques that use symbolic execution for equivalence checking and software equivalence checking techniques that are based on other, related, approaches.

Symbolic execution. DSE [38] and IMP-S [6] are the closest to our work. They are extensively discussed in Section 1 and compared with our tool. In a nutshell, our work extends DSE with an abstraction-refinement loop and a set of heuristics for selecting preferred refinement candidates. Our work is orthogonal to IMP-S as our proposed heuristics focus on selecting the best refinement candidate based on the structure of the symbolic summaries and structure of the code. In fact, a fruitful direction of possible future work could be applying our proposed heuristics over IMP-S, which could result in a solution that better handles the non-equivalent cases. We also intend to explore broader solutions for non-equivalent cases as a part of the future work.

ModDiff [45] is a modular and demand-driven analysis which performs a bottom-up summarization of methods common between versions and only refines the paths of the methods that are needed to prove equivalence. CLEVER [34] formulates the notion of client-specific equivalence checking and develops an automated technique optimized for checking the equivalence of downstream components by leveraging the fact that their calling context is unchanged. As such, CLEVER only explores paths that are relevant within the client context. Both these techniques scale the analysis to work on the inter-method level and only consider full-path pruning at the individual method level. Our work is thus orthogonal and complementary to these approaches; combining the approaches could be explored in future work.

Model checking and theorem proving. SymDiff [31] checks the equivalence of two methods given their behavioral specification provided by the user. RVT [26] proves partial equivalence of two related programs, showing that they produce the same outputs for all inputs on which they terminate, according to a set of proof rules. Recursive calls are first abstracted as uninterpreted functions, and then the proof rules for non-recursive functions are applied in a bottom-up fashion. Our technique does not rely on any user-defined rule. Also, while these approaches can only prove the equivalence, our technique can formally prove non-equivalence.

Relational verification approaches, e.g., [12, 22, 35, 39, 43], focus on verifying properties about two programs or two runs of the same program, including program equivalence. These approaches mostly focus on loopy and/or recursive programs; they aim to align programs and/or their execution traces to a so-called *product program* and then identify invariants at the alignment points, effectively transforming a verification problem into an invariant synthesis task. Our work is orthogonal to these approach as it does not need to discover invariants but rather assumes that the two compared versions are closely related, allowing for better scalability of our tool. Yet, combining and comparing our technique with these approaches could be an interesting topic of possible future work.

Incremental verification. This line of work aims to reuse results from prior verification as programs evolve, assuming that properties of a client to be verified are given. For example, Sery et al. [42] uses a lazy approach, implemented in a tool named eVolCheck, which extracts the property-directed summaries of all function calls (i.e., that capture only the relevant information needed to verify the assertion), and then locally validates the old summaries w.r.t. the updated version of the program, thus effectively avoiding the need to re-verify the updated code whenever possible. Chaki et al. [11] use state machine abstractions to analyze whether every behavior that should be preserved is still available and whether added behaviors conform to their respectful properties. Fedyukovich et al. [21] offer an incremental verification technique for checking equivalence w.r.t. program properties designed specifically for loopy programs. Our work does not rely on verification results from previous versions and does not require any user-generated specifications.

Concolic execution. Shadow symbolic execution [10, 36] uses a combination of concrete and symbolic runs to identify path divergence between subsequent program versions. The goal of this technique is to generate an input that will make two versions of the program take a different path. However, it cannot prove or disprove functional equivalence. UC-KLEE [41] is an equivalence checking tool for C programs built on top of the symbolic engine KLEE [9]. It automatically synthesizes inputs and verifies that they produce equivalent outputs on a finite number of paths. Yet, this tool cannot prove or disprove equivalence in full. While all these techniques mostly aim at identifying examples to demonstrate differences, the main focus of our work is on formally proving equivalence.

7 CONCLUSION

In this paper, we proposed an iterative symbolic-execution-based approach for checking equivalence of two versions of a method. It leverages the idea that versions share a large portion of common code, which is not necessarily required to prove equivalence and can be abstracted away using uninterpreted functions. Such abstraction helps “hide” complex parts of the code, such as non-linear arithmetic and unbounded loops, that lead to ‘unknown’ or imprecise results. The key contribution of our approach lies in identifying the set of common code statements that can be abstracted away vs. common code statements that are needed for establishing equivalence. We developed a set of heuristics that help to distinguish between such cases and evaluated both the contributions of individual heuristics and of their composition, comparing our tool with two state-of-the-art method-level equivalence checking techniques: DSE and IMP-S.

For the evaluation, we used the existing equivalence checking benchmarks proposed by earlier work and also devised a more complete set of benchmarks that contains realistic methods with complex non-linear arithmetic operations borrowed from the field of scientific computing. The results of our evaluation show that our tool substantially outperforms existing approaches for proving equivalence and performs comparably when applied to non-equivalent cases. The implementation of our approach, the benchmarks we developed, and our experimental evaluation results are available online [5].

Acknowledgments. We thank the authors of Java PathFinder for their prompt help resolving some of the issues that we had. 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

[1] [n.d.]. Github. https://github.com.

[2] [n.d.]. Refactoring GURU. https://refactoring.guru/remove-assignments-to-parameters.

[3] 2019. ASM-DefUse. *Software Analysis Experimentation Group* (2019). https://github.com/saeg/asm-defuse.

[4] 2020. *ARDiff. https://www.doi.org/10.17605/OSF.IO/CHM2K*.

[5] 2020. *Supplementary Materials. https://resess.github.io/PaperAppendices/ARDiff/*.

[6] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries. In *Proc. of SPIN Workshop on Model Checking of Software*.

[7] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a Code Manipulation Tool to Implement Adaptable Systems. *Adaptable and Extensible Component Systems* 30, 19 (2002).

[8] Jerry R Burch and David L Dill. 1994. Automatic Verification of Pipelined Microprocessor Control. In *Proc. of the International Conference on Computer Aided Verification (CAV)*. 68–80.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*. 209–224.

[10] Cristian Cadar and Hristina Palikareva. 2014. Shadow Symbolic Execution for Better Testing of Evolving Software. In *Proc. of the International Conference on Software Engineering (ICSE)*. 432–435.

[11] Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. 2008. Verification of Evolving Software via Component Substitutability Analysis. In *Formal Methods in System Design*. 235–266.

[12] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proc. of Programming Language Design and Implementation (PLDI)*.

[13] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided Abstraction Refinement. In *Proc. of the International Conference on Computer Aided Verification (CAV)*. 154–169.

[14] Stephen A Cook. 1971. The Complexity of Theorem-proving Procedures. In *Proc. of the Symposium on Theory of Computing (STOC)*.

[15] Martin Davis. 1973. Hilbert's Tenth Problem is Unsolvable. *The American Mathematical Monthly* 80, 3 (1973), 233–269.

[16] Martin Davis. 1973. Hilbert's Tenth Problem is Unsolvable. *The American Mathematical Monthly* 80, 3 (1973), 233–269.

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340.

[18] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (2011), 69–77.

[19] Peter Dinges and Gul Agha. 2014. Solving Complex Path Conditions Through Heuristic Search on Induced Polytopes. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*. 425–436.

[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 313–324.

[21] Grigory Fedyukovich, Arie Gurfinkel, and Nastasha Sharygina. 2016. Property-Directed Equivalence via Abstract Simulation. In *Proc. of the International Conference on Computer Aided Verification*. 433–453.

[22] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 349–360.

[23] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[24] M. Fowler. [n.d.]. Refactoring home page. https://refactoring.com/catalog/.

[25] Benny Godlin and Ofer Strichman. 2008. Inference Rules for Proving the Equivalence of Recursive Procedures. *Acta Informatica* 45, 6 (2008), 403–439.

[26] Benny Godlin and Ofer Strichman. 2013. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability* 23, 3 (2013), 241–258.

[27] Joseph Y Halpern. 1991. Presburger Arithmetic with Unary Predicates is Π 1-1 Complete. *The Journal of Symbolic Logic* 56, 2 (1991), 637–642.

[28] James C King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[29] Daniel Kroening and Ofer Strichman. 2008. Equality Logic and Uninterpreted Functions. In *Decision Procedures*. 59–80.

[30] Andreas Kuehlmann and Florian Krohm. 1997. Equivalence Checking Using Cuts and Heaps. In *Proc. of the Design Automation Conference (DAC)*. 263–268.

[31] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A Language-agnostic Semantic Diff Tool for Imperative Programs. In *Proc. of the International Conference on Computer Aided Verification (CAD)*. 712–717.

[32] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning-Based Constraint Solving. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 554–559.

[33] Yu-Seung Ma and Jeff Offutt. [n.d.]. Description of muJava's Method-level Mutation Operators. ([n. d.]).

[34] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific Equivalence Checking. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 441–451.

[35] Dmitry Mordvinov and Grigory Fedyukovich. 2019. Property Directed Inference of Relational Invariants. In *Proc. of Formal Methods in Computer Aided Design (FMCAD)*. 152–160.

[36] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences Between Software Versions. In *Proc. of the International Conference on Software Engineering (ICSE)*. 1181–1192.

[37] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java bytecode. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 179–180.

[38] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. 2008. Differential Symbolic Execution. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*.

[39] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. 2018. Exploiting Synchrony and Symmetry in Relational Verification. In *Proc. of International Conference on Computer Aided Verification (CAV)*. 164–182.

[40] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*.

[41] David A Ramos and Dawson R Engler. 2011. Practical, Low-effort Equivalence Verification of Real Code. In *Proc. of the International Conference on Computer Aided Verification (CAV)*. 669–685.

[42] Ondrej Sery, Girogory Fedyukovich, and Natasha Sharygina. 2012. Incremental Upgrade Checking by Means of Interpolation-Based Function Summaries. In *Proc. of the Formal Methods in Computer-Aided Design Conference*. 114–427.

[43] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property Directed Self Composition. In *Proc. of International Conference on Computer Aided Verification (CAV)*. 161–179.

[44] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S Păsăreanu. 2011. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proc. of NASA Formal Methods Symposium*. 359–374.

[45] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular Demand-driven Analysis of Semantic Difference for Program Versions. In *Proc. of the International Static Analysis Symposium (SAS)*. 405–427.

[46] Christopher A Welty. 1997. Augmenting Abstract Syntax Trees for Program Understanding. In *Proc. of the International Conference Automated Software Engineering (ICSE)*. 126–133.

[47] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. 2011. Precise Identification of Problems for Structural Test Generation. In *Proc. of the International Conference on Software Engineering (ICSE)*. 611–620.