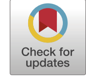
The Journal of Systems and Software 213 (2024) 112037

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



PASDA: A partition-based semantic differencing approach with best effort classification of undecided cases ☆

Johann Glock [a,*], Josef Pichler [b], Martin Pinzger [a]

[a] Department of Informatics Systems, University of Klagenfurt, Universitätsstraße 65-67, Klagenfurt, 9020, Austria

[b] School of Informatics, Communications and Media, University of Applied Sciences Upper Austria, Softwarepark 11, Hagenberg im Mühlkreis, 4232, Austria

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.7595851

Keywords:
Equivalence checking
Program analysis
Symbolic execution

ABSTRACT

Equivalence checking is used to verify whether two programs produce equivalent outputs when given equivalent inputs. Research in this field mainly focused on improving equivalence checking accuracy and runtime performance. However, for program pairs that cannot be proven to be either equivalent or non-equivalent, existing approaches only report a classification result of *unknown*, which provides no information regarding the programs' non-/equivalence.

In this paper, we introduce PASDA, our partition-based semantic differencing approach with best effort classification of undecided cases. While PASDA aims to formally prove non-/equivalence of analyzed program pairs using a variant of differential symbolic execution, its main novelty lies in its handling of cases for which no formal non-/equivalence proof can be found. For such cases, PASDA provides a best effort equivalence classification based on a set of classification heuristics.

We evaluated PASDA with an existing benchmark consisting of 141 non-/equivalent program pairs. PASDA correctly classified 61%–74% of these cases at timeouts from 10 s to 3600 s. Thus, PASDA achieved equivalence checking accuracies that are 3%–7% higher than the best results achieved by three existing tools. Furthermore, PASDA's best effort classifications were correct for 70%–75% of equivalent and 55%–85% of non-equivalent cases across the different timeouts.

1. Introduction

In the context of software programs, the goal of functional equivalence checking is to verify whether two programs produce equivalent outputs when given equivalent inputs (Godlin and Strichman, 2009). This information can be used, for example, to verify compiler optimizations (Dahiya and Bansal, 2017), to provide assurance that refactorings do not introduce unintended functional changes (Person et al., 2008), to check whether changes in libraries affect their clients (Mora et al., 2018), or even for security related analyses such as distinguishing benign integer overflows from harmful ones (Sun et al., 2016) and classifying which malware family a given program belongs to (Mercaldo and Santone, 2021).

Classification accuracy and runtime performance of equivalence checking approaches have seen continuous improvements throughout the years (Person et al., 2011; Backes et al., 2013; Felsing et al., 2014; Jakobs and Wiesner, 2022), which has enabled these approaches to provide non-/equivalence proofs for more complex programs in shorter amounts of time. However, for cases that cannot be proven to be either equivalent or non-equivalent, existing approaches such as UC-KLEE (Ramos and Engler, 2011), PEQcheck (Jakobs, 2021), and ARDiff (Badihi et al., 2020) only output a classification of Unknown, which provides no indication about the potential non-/equivalence of the two analyzed programs. For example, ARDiff classifies the two equivalent programs shown in Listing 1 as well as the two non-equivalent programs shown in Listing 2 as Unknown, but does not provide any further information beyond this.

In this paper, we introduce PASDA, our partition-based semantic differencing approach with best effort classification of undecided cases. While PASDA aims to formally prove non-/equivalence of analyzed program pairs, its main novelty lies in its handling of cases for which no such formal proof can be found. Instead of classifying all such cases as Unknown, PASDA reports a best effort equivalence classification of either Maybe_Eq or Maybe_Neq if it can provide at least *part of* a non-/equivalence proof on either the program or partition level. In this way, PASDA offers a more complete description of program behaviors than existing approaches since it reduces the part of the input space for which no information is provided.

☆ Editor: Burak Turhan.

* Corresponding author.
E-mail address: johann.glock@aau.at (J. Glock).

https://doi.org/10.1016/j.jss.2024.112037
Received 17 October 2023; Received in revised form 7 February 2024; Accepted 22 March 2024
Available online 28 March 2024
0164-1212/© 2024 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

```
1  double eq_v1(int x, double y) {
2    for (int i = 0; i < 1; i++) { x += 0; }
3    return x + y + 1;
4  }
5
6  double eq_v2(int x, double y) {
7    for (int i = 0; i < x; i++) { x *= 1; }
8    return 1 + x + y;
9  }
```

Listing 1: Example of two equivalent programs.

```
1  double neq_v1(double x) {
2    if (x <= 0) { return 1; }
3    if (x >  9) { return Math.tan(1 * x); }
4    return Math.tan(1 * x) < 0 ? -1 : 0;
5  }
6
7  double neq_v2(double x) {
8    if (x <= 0) { return 2; }
9    if (x >  9) { return Math.tan(2 * x); }
10   return Math.tan(2 * x) < 0 ? -1 : 0;
11 }
```

Listing 2: Example of two non-equivalent programs.

For example, PASDA classifies the two equivalent programs in Listing 1 as MAYBE_EQ rather than UNKNOWN because it can provide equivalence proofs for all partitions up to a pre-configured depth limit for loop exploration. For the two non-equivalent programs in Listing 2, PASDA can even provide an overall non-equivalence proof, thus classifying the two programs as NEQ rather than UNKNOWN. In this case, the eight identified partitions (see Table 3) of the non-equivalent programs are classified as either EQ, NEQ, MAYBE_NEQ, or UNKNOWN depending on whether (i) both, (ii) one of, or (iii) none of reachability and output non-/equivalence can be proven.

To further improve the utility of its outputs, PASDA does not only process this partition-level information internally, but also reports it as supporting information alongside its program-level equivalence classification results. Specifically, for each input partition that PASDA identifies during a symbolic execution (King, 1976) of the two target programs, it aims to collect (a) concrete and symbolic input and output values, (b) a partition-level equivalence classification, and (c) the lines of code that are covered by the corresponding execution path. Such information has been found to benefit developers' understanding of program analysis results (LaToza and Myers, 2010; Parnin and Orso, 2011; Winter et al., 2022). Similar information is provided by some existing equivalence checking approaches such as DSE (Person et al., 2008), SymDiff (Lahiri et al., 2012), and PRV (Böhme et al., 2013).

Our implementation of PASDA takes the source code of two Java programs as its input and first constructs a product program (Barthe et al., 2011; Beckert and Ulbrich, 2018) that combines the behavior of these two target programs. PASDA then uses the symbolic execution engine Symbolic PathFinder (Păsăreanu and Rungta, 2010) to symbolically execute the product program. For each partition identified during the symbolic execution, PASDA collects the corresponding path condition and symbolic output values of the two target programs. This information is sent to the theorem prover Z3 (De Moura and Bjørner, 2008) to produce an equivalence classification for each partition. Once all partitions have been analyzed, PASDA aggregates the *partition-level* results to produce a *program-level* result. If the program-level result is a proof of non-/equivalence, PASDA reports this as the final result of its analysis. Otherwise, PASDA employs iterative abstraction and refinement of unchanged parts of the source code (Badihi et al., 2020) to conduct further analysis iterations until a non-/equivalence proof is found or a given timeout is reached.

We evaluated the equivalence checking accuracy and runtime performance of PASDA with an existing benchmark (Badihi et al., 2020) consisting of 73 equivalent and 68 non-equivalent program pairs and compared the results to the three existing tools ARDIFF (Badihi et al., 2020), DSE (Person et al., 2008), and PRV (Böhme et al., 2013). PASDA correctly classified 61%–74% of analyzed cases at six different timeout settings ranging from 10 seconds (s) to 3600 s. In our evaluation, PASDA's classification accuracy is, therefore, 3%–7% higher than those that we observed for the three existing tools, which correctly classified at most 58% of cases at the 10 s timeout setting and at most 67% of cases at the 3600 s timeout setting. Furthermore, PASDA's best effort classifications for undecided cases were correct for 70%–75% of equivalent cases and 55%–85% of non-equivalent cases across the six analyzed timeout settings.

We envision that PASDA's best effort equivalence classifications and corresponding partial non-/equivalence proofs will benefit use cases such as test case prioritization (Khatibsyarbini et al., 2018), fault localization (Wong et al., 2016), and general support for manual debugging tasks. All of these use cases benefit from more complete information about non-/equivalent program behaviors even without full non-/equivalence proofs. While an in-depth analysis of such use cases is beyond the scope of this paper, we plan to investigate them in our future work.

The contributions that we make in this paper are as follows:

1. PASDA, an equivalence checking approach for software programs that provides best effort classifications for cases that cannot be proven to be non-/equivalent,
2. an evaluation of PASDA's equivalence checking accuracy and runtime performance by comparing it to three state-of-the-art equivalence checking tools using an established equivalence checking benchmark,
3. an evaluation of PASDA's best effort classification results,
4. a publicly available implementation of PASDA that supports equivalence checking of Java programs.

Throughout the remainder of this paper, we first present background information on the use of symbolic execution for equivalence checking in Section 2. In Section 3, we describe PASDA and evaluate it in Section 4. Section 5 discusses the benefits and potential use cases of our approach as well as threats to the validity of our results. We present related work in Section 6 and draw the conclusions in Section 7.

2. Background

This section introduces relevant terminology that we use throughout the rest of this paper to describe our approach. Furthermore, it provides descriptions of the three equivalence checking approaches DSE (Person et al., 2008), ARDIFF (Badihi et al., 2020), and PRV (Böhme et al., 2013) which originally introduced key ideas (differential symbolic execution, iterative abstraction and refinement, and partition-based regression verification) that we reuse in our approach.

2.1. Symbolic execution

Symbolic execution is a technique for systematically exploring all possible execution paths of a given program (Baldoni et al., 2018; Cadar and Sen, 2013). The main ideas employed by symbolic execution are to (i) use symbolic rather than concrete values to represent the program state and to (ii) follow all reachable execution paths whenever a branching point is encountered during the execution (King, 1976). For each execution path, symbolic execution collects a *path condition* that contains the constraints that have to be satisfied by the input variables

to reach the path, and the outputs produced by following the path, which are represented as a function of the input variables (Păsăreanu and Visser, 2009). The input values that satisfy a given path condition are commonly referred to as an *(input) partition*.

Once all reachable paths have been explored, symbolic execution outputs a *program summary* which consists of a disjunction of one or more *partition-effects pairs* (Person et al., 2008). Each partition-effects pair consists of a conjunction of the path condition and corresponding outputs of a single execution path. Unreachable paths, i.e., paths that have unsatisfiable path conditions, are skipped during the symbolic execution and not included in the program summaries. For example, the program summary that is produced by a symbolic execution of the program `neq_v2` in Listing 2 is the following:

```
1       x<=0                                ∧ RET=2
2   ∨ !(x<=0) ∧   x>9                       ∧ RET=tan(2*x)
3   ∨ !(x<=0) ∧ !(x>9) ∧   tan(2*x)<0       ∧ RET=-1
4   ∨ !(x<=0) ∧ !(x>9) ∧ !(tan(2*x)<0)      ∧ RET=0
```

Limitations of symbolic execution include its high computational cost, its inability to produce complete summaries in the presence of unbounded loops and recursion, and its inability to handle complex expressions (e.g., non-linear arithmetic) which are intractable for modern decision procedures (Person et al., 2008; Cadar et al., 2011; Cadar and Sen, 2013).

2.2. Differential symbolic execution

Differential symbolic execution (DSE) uses symbolic execution to characterize differences in the input–output behavior of two programs (Person et al., 2008). More specifically, DSE first collects summaries of the two target programs via symbolic execution. It then compares these summaries to identify inputs for which the two programs produce different outputs. If no differences in the input–output behavior are identified, the two programs are equivalent and DSE does not produce any further output. However, if the two programs are non-equivalent, DSE outputs a *behavioral delta* that consists of the non-equivalent input partitions and corresponding outputs that are produced for these inputs by the two target programs.

To avoid some of the limitations of symbolic execution, DSE replaces parts of the source code that are unchanged across the two target programs with calls to uninterpreted functions (UIFs) before running the symbolic execution. For example, assuming that line 2 of `eq_v1` is syntactically equivalent to line 7 of `eq_v2` in Listing 1, both lines would be replaced with a UIF call $x = UIF(x)$. While this abstraction would lead to an overapproximation of actual program behavior, it would enable DSE to prove the two programs to be equivalent without having to bound the number of loop iterations that are analyzed. Similar benefits can be achieved when abstracting program constructs such as recursion and non-linear arithmetic that are also difficult for symbolic execution to handle.

One of the main limitations of DSE is that it needs to *fully* analyze both target programs before an equivalence classification can be produced. This makes it unsuitable for use cases where its execution time exceeds given time constraints. Furthermore, DSE exhibits lower equivalence checking accuracy than more recent tools such as ARDiff (Badihi et al., 2020). Finally, even though DSE produces behavioral deltas for Neq partitions, it does not provide input–output descriptions for partitions classified as Eq or Unknown. For example, DSE classifies the programs in Listing 1 as Unknown due to depth-limiting, and Listing 2 as Unknown due to the presence of the `tan()` function (which generally is not fully modeled by modern decision procedures), but does not provide any information beyond this.

2.3. Iterative abstraction and refinement

The goal of ARDiff (Badihi et al., 2020) is to reduce the number of programs that cannot be provably classified as non-/equivalent due to the introduction of UIFs while preserving the benefits that this abstraction provides. ARDiff accomplishes this through an iterative process that starts with the same abstraction of unchanged source code that DSE uses. If no non-/equivalence proof can be found at this level of abstraction, ARDiff refines one of the UIFs, replacing it with the original unchanged source code. The resulting partially refined program is again checked for equivalence, and further refinement iterations are conducted until non-/equivalence can be proven, no further refinement is possible, or a timeout is reached.

To choose which UIF should be refined in each iteration, ARDiff employs three heuristics. Heuristic 1 (H1) marks those UIFs as refinement candidates for which an assignment exists that enables a non-/equivalence proof irrespective of the assignments of all other UIFs. Heuristic 2 (H2) marks those UIFs as refinement candidates that occur a different number of times in the two programs. Heuristic 3 (H3) then ranks the refinement candidates according to how deeply they are nested in loops and how many non-linear arithmetic operations each candidate replaces. If no refinement candidates are identified by H1 and H2, all UIFs are included in H3's ranking. The UIF with the lowest rank is refined. We refer the reader to Badihi et al. (2020) for a more detailed description of the three heuristics.

While ARDiff achieves better equivalence checking accuracy than DSE, this comes at the cost of longer runtimes. Furthermore, ARDiff inherits DSE's requirement that the two target programs have to be *fully* analyzed before an equivalence classification can be provided. Finally, ARDiff does *not* preserve DSE's behavioral deltas as a description of non-equivalent program behaviors — it only produces program-level equivalence classifications. For example, ARDiff reports the two cases shown in Listings 1 and 2 as Unknown for the same reasons as DSE, but does not provide any other information beyond these equivalence classifications.

2.4. Partition-based regression verification

DSE and ARDiff both need to fully analyze the two target programs before a non-/equivalence proof can be provided. Partition-based regression verification (PRV), on the other hand, can provide non-equivalence proofs for many cases after only a partial analysis of the target programs (Böhme et al., 2013). PRV accomplishes this by checking non-/equivalence on the partition level instead of the program level. With this approach, two programs are proven to be equivalent if *all* partitions can be proven to be so, which still requires a full analysis. However, two programs are proven to be non-equivalent as soon as the *first* input partition for which the two target programs produce different outputs is identified.

PRV's analysis starts by identifying a random assignment of input values from the common input space of the two target programs. These inputs are used to execute both programs, which provides the program outputs and the corresponding path conditions. Outputs are checked for non-/equivalence, thus marking the partition as either Eq or Neq if a non-/equivalence proof is found, or Unknown otherwise. Exploration then continues with a new set of input values from the unexplored part of the common input space. The process repeats until the full input space has been explored or some timeout is reached.

There are two main benefits that PRV's partition-based approach provides: (i) non-equivalence of two programs can be proven without a full analysis of all program paths, and (ii) even if the analysis is interrupted ahead of time, non-/equivalence proofs for all explored partitions are preserved. However, PRV does not employ UIFs to abstract unchanged parts of the source code and, therefore, cannot benefit from the advantages that such an abstraction entails (see Section 2.2).

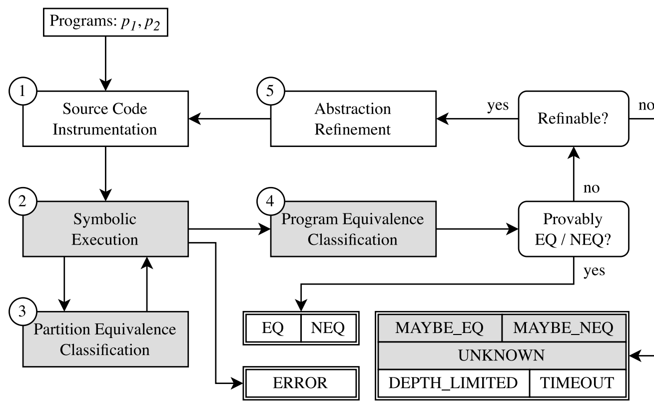


Fig. 1. Overview of PASDA's semantic differencing process. Elements with a white background are identical to the state-of-the-art, whereas elements with a gray background are modified or newly introduced by PASDA.

Furthermore, PRV still distinguishes only three equivalence classifications (Eq, Neq, and Unknown) at the partition level. For example, PRV classifies all partitions in Listing 2 that return via line 4 in `neq_v1` and line 10 in `neq_v2` as Unknown because it cannot generate inputs that satisfy the corresponding path conditions (due to the presence of the `tan()` function), but does not provide any further information beyond the Unknown equivalence classification for these partitions.

3. The PASDA approach

Our approach, which we refer to as PASDA, combines ideas from DSE (Person et al., 2008) (abstraction of unchanged code parts), ARDiff (Badihi et al., 2020) (iterative abstraction and refinement), and PRV (Böhme et al., 2013) (partition-based analysis) and further extends them with a best effort classification of programs and partitions that cannot be formally proven to be either equivalent or non-equivalent.

An overview of the full approach is shown in Fig. 1. PASDA starts with a source code instrumentation step that creates a product program which combines the two target programs p_1 and p_2. Starting at PASDA's second iteration, this step also replaces unchanged parts of the source code of the two target programs with UIFs. Once the instrumentation is complete, PASDA symbolically executes the product program to collect information for equivalence checking. If non-/equivalence cannot be proven based on the collected information, the three heuristics proposed by Badihi et al. (2020) are used to choose a UIF to replace with the original unchanged code, thereby creating two new programs that are again checked for non-/equivalence. This iterative refinement process continues until one of the following conditions holds: (i) equivalence or non-equivalence can be proven, (ii) no further refinement is possible, or (iii) the runtime exceeds the configured timeout.

The main difference that sets PASDA apart from existing equivalence checking approaches is how it handles limitations of the used decision procedures. For example, if a program path cannot be proven to be reachable or unreachable, or if the outputs of two programs cannot be proven to be equivalent or non-equivalent, existing tools simply classify the corresponding programs or partitions as Unknown. PASDA, on the other hand, takes a more differentiated view. Rather than classifying all such cases as Unknown, it distinguishes between Maybe_Eq, Maybe_Neq, and Unknown results based on a set of classification heuristics (see Step 3 and Step 4). Furthermore, PASDA provides not only program- and partition-level equivalence classifications, but also aims to provide execution traces as well as concrete and symbolic inputs and outputs for identified partitions (see Step 2 and Section 3.6) since such information has been found to benefit developers' understanding of program analysis results (LaToza and Myers, 2010; Parnin and Orso, 2011; Winter et al., 2022).

```
1  public class ProductProgram {
2    ...
3
4    public static void main(String[] args) {
5      ProductProgram.run(${values});
6    }
7
8    public static ${type} run(${params}) {
9      ${type} result1 = ${defaultValue};
10     ${type} result2 = ${defaultValue};
11     Throwable error1 = null;
12     Throwable error2 = null;
13
14     try {
15       result1 = ${cls1}.${method1}(${vars});
16     } catch (Throwable e) {
17       error1 = e;
18     }
19     try {
20       result2 = ${cls2}.${method2}(${vars});
21     } catch (Throwable e) {
22       error2 = e;
23     }
24
25     checkEquivalence(error1, error2);
26     checkEquivalence(result1, result2);
27
28     return result2;
29   }
30
31   public static void checkEquivalence(...)
32     throws DifferentOutputsException { ... }
33 }
```

Listing 3: The product program template used by PASDA.

In the following, we provide more detailed descriptions of PASDA's five processing steps (i.e., source code instrumentation, symbolic execution, partition equivalence classification, program equivalence classification, and abstraction refinement) in Sections 3.1–3.5. A description of the outputs that PASDA produces is then provided in Section 3.6.

3.1. Step 1: Source code instrumentation

At the start of every iteration, PASDA constructs a product program (Barthe et al., 2011; Beckert and Ulbrich, 2018) that enables it to check the equivalence of the two target programs in a single symbolic execution run (Lahiri et al., 2012; Ramos and Engler, 2011). Furthermore, starting at its second iteration, PASDA replaces parts of the source code that are unchanged across the two target programs with UIFs (Person et al., 2008).

Construction of the product program. Listing 3 shows the template that is used by PASDA to construct the product program. The product program first executes the two given target programs (lines 15 and 20) and then checks whether their effects, i.e., return values or thrown exceptions, are equivalent (lines 25 and 26). If the effects are *not* equivalent, a `DifferentOutputsException` is thrown. Otherwise, the result produced by the two programs is returned as the output of the product program (line 28).

For the product program template to be applicable to a given pair of target programs, the following assumptions must hold: (i) the two programs must take the same number of input arguments, and the types of the input arguments as well as the output types of the two programs must match, (ii) all effects produced by the two target programs must be observable via return values or raised exceptions, and (iii) the two

target programs must both be deterministic, i.e., both programs must always produce equivalent effects when called with equivalent inputs.

Assumptions (i) and (ii) could potentially be lifted through a more refined preprocessing approach. More specfically, program pairs that do not satisfy assumption (i) could be detected and classified as non-equivalent through a lightweight preprocessing step that compares the input and output types of the two target programs. Furthermore, programs that do not satisfy assumption (ii) could be transformed to include side effects (e.g., modifications of global variables, console outputs, etc.) in the outputs of the programs prior to equivalence checking. We leave such improvements for future work.

Introduction of UIFs. Starting at its second iteration, PASDA uses instrumented variants of the two target programs as the targets of its analysis. In these instrumented programs, parts of the source code that are syntactically unchanged across the two original programs are replaced with UIFs. We reuse ARDIFF's implementation (Badihi et al., 2022) of this instrumentation step, which uses GumTree (Falleri et al., 2014) to identify unchanged parts of the source code. When applied to our examples in Listings 1 and 2, none of the statements are replaced, because each line of code contains a syntactic change.

3.2. Step 2: Symbolic execution

PASDA uses Symbolic PathFinder (SPF) (Păsăreanu and Rungta, 2010) to symbolically execute the product program created during the instrumentation step. For each identified path that is either provably or maybe reachable (for further details on reachability, see *partition reachability classification* in Section 3.3), PASDA aims to collect the following information:

(a) one partition-effects pair for each program version,
(b) equivalence information for the two partition-effects pairs,
(c) information about the lines of code covered by the path.

The (a) partition-effects pairs and (c) coverage information are collected via listeners that hook into the lifecycle events of SPF. For example, coverage information is collected by storing the current line number whenever SPF raises an `instructionExecuted` event. The (b) equivalence information is calculated whenever a `checkEquivalence` call is reached in lines 25 and 26 of the product program shown in Listing 3. For exceptions, PASDA only checks whether their types are the same to determine non-/equivalence. For regular return values, PASDA performs the partition equivalence classification described in Step 3. Note that the full data for (a)–(c) can only be collected for paths that (i) are (maybe) reachable, and (ii) are not depth-limited. In the following paragraphs, we describe which parts of the data are collected for the remaining paths that do not satisfy these criteria.

Unreachable paths. A path is said to be unreachable if its path condition is provably unsatisfiable. For example, in `eq_v1` in Listing 1, all paths that perform more than a single loop iteration are unreachable because the looping condition $i < 1$ cannot be satisfied anymore once i has been incremented at the end of the first loop iteration. Since unreachable paths represent program flow that cannot occur in practice, all unreachable paths are skipped during the symbolic execution, and PASDA does not collect any of the data listed in (a)–(c) for them.

Depth-limited paths. For paths that are not fully analyzed due to the configured depth limit, PASDA only collects partial data for (a)–(c). Specifically, it collects (a') the path condition of the path, and (c') the covered lines of code up to the point where the depth limit was reached. For example, when `eq_v2` from Listing 1 is symbolically executed with a depth limit setting of 10, all inputs $x > 10$ produce depth-limited paths. For these inputs, execution proceeds as normal until the program is about to enter the 11th loop iteration. At this point, symbolic execution of the current path stops with a path condition of $x > 10$, having covered only line 2. No (b) equivalence information can be collected because no return value has been produced at this point. The corresponding partition is classified as DEPTH_LIMITED and the symbolic execution then proceeds as usual for the remaining paths that have not been explored yet.

PASDA alternates between symbolic execution (Step 2) and partition equivalence classification (Step 3) until all (maybe) reachable paths have been explored and checked for non-/equivalence. The process then continues with program equivalence classification (Step 4).

3.3. Step 3: Partition equivalence classification

As described in the previous section, the values returned by the two target programs are checked for equivalence whenever the symbolic execution reaches the `checkEquivalence` call in line 26 of the product program shown in Listing 3. Two factors are combined to produce the overall partition-level equivalence classification: partition reachability and partition output equivalence. In this section, we describe how these factors are calculated and how the overall partition-level equivalence classification is derived from them.

Partition reachability classification. Reachability can be classified as REACHABLE, MAYBE_REACHABLE, or UNREACHABLE. Classification is influenced by two factors: (i) the presence of UIFs in the path condition of the partition (UIF = yes/no), and (ii) the result of a Z3 query that checks whether the path condition is satisfiable (PC-Query = SAT/UNSAT/UNKNOWN). An enumeration of the possible combinations of these factors and their corresponding classifications are shown in Table 1. For example, paths in `neq_v2` from Listing 2 that return via line 8 or 9 are REACHABLE. Paths that return via line 10 are MAYBE_REACHABLE because Z3 cannot prove whether $Math.tan(2 * x) < 0$ holds for all x. Paths in `eq_v1` from Listing 1 that perform more than one loop iteration are UNREACHABLE. As described in the previous section, UNREACHABLE partitions are skipped by the symbolic execution and, therefore, do not have to be considered during partition equivalence classification.

Partition output equivalence classification. Output equivalence can be classified as EQ, NEQ, MAYBE_EQ, MAYBE_NEQ, or UNKNOWN. Classification is influenced by three factors: (i) the presence of UIFs in the partition-effects pairs of the current path (UIF = yes/no), (ii) the result of a Z3 query that checks whether the outputs of the two programs are non-equivalent (NEQ-Query = SAT/UNSAT/UNKNOWN), and (iii) the result of a Z3 query that checks whether the outputs of the two programs are equivalent (EQ-Query = SAT/UNSAT/UNKNOWN). An overview of the possible combinations of these factors and their corresponding classifications are shown in Table 2. For example, the path that returns via line 2 in `neq_v1` and line 8 in `neq_v2` in Listing 2 is classified as NEQ, whereas the path that returns via line 3 in `neq_v1` and line 9 in `neq_v2` is classified as UNKNOWN because Z3 cannot prove whether or not $Math.tan(1 * x)$ and $Math.tan(2 * x)$ are equivalent for all x.

Overall partition equivalence classification. The partition reachability classification and partition output classification are combined to produce the overall partition-level equivalence classification. Specifically, a reachability result of MAYBE_REACHABLE turns NEQ results into MAYBE_NEQ results (because in this case, it is not possible to prove that the non-equivalence can be observed in a concrete execution of the two original programs). For all other combinations, the overall classification is the same as the output classification. Thus, each partition is assigned to one of six equivalence classes: EQ, NEQ, MAYBE_EQ, MAYBE_NEQ or UNKNOWN if the partition is *not* depth-limited (as described in this section), or DEPTH_LIMITED otherwise (as described in the previous section). Classification results are returned to Step 2 which then continues with the analysis of the remaining paths of the product program.

Table 1
Partition reachability classification.

Uif	Pc-Query	Reachability classification
No	Sat	Reachable
No	Unsat	Unreachable
No	Unknown	Maybe_Reachable
Yes	Sat	Maybe_Reachable
Yes	Unsat	Unreachable
Yes	Unknown	Maybe_Reachable

Table 2
Partition output equivalence classification.

Uif	Neq-Query	Eq-Query	Output classification
*	Unsat	*	Eq
*	Unknown	Sat	Maybe_Eq
*	Unknown	Unsat	Neq
*	Unknown	Unknown	Unknown
No	Sat	*	Neq
Yes	Sat	Sat	Maybe_Neq
Yes	Sat	Unsat	Neq
Yes	Sat	Unknown	Maybe_Neq

3.4. Step 4: Program equivalence classification

Once the symbolic execution concludes, the results of the partition-level equivalence classifications are aggregated to produce the program-level equivalence classification of the current iteration. A program-level result of Eq is reported if all partitions could be identified, analyzed, and proven to be Eq within the given timeout. A result of Maybe_Eq is reported if at least some partitions were identified and classified as Eq or Maybe_Eq but none were classified as Neq or Maybe_Neq. Otherwise, the program-level result is determined by the partition-level result with the highest priority, where priorities are as follows: Neq > Maybe_Neq > Unknown > Depth_Limited. For example, Table 3 lists the partition-level data obtained by PASDA for `neq_v1` and `neq_v2` in Listing 2. Because at least one partition is classified as Neq, the two programs are classified as Neq. Special handling exists to provide classifications in the presence of errors and timeouts. If an error (e.g., `OutOfMemoryError`) occurs during PASDA's execution, a classification of Error is reported. If PASDA cannot analyze even a single partition within the given timeout, a classification of Timeout is reported.

3.5. Step 5: Abstraction refinement

Abstraction refinement takes place at the end of an iteration if neither equivalence nor non-equivalence can be proven. Refinement, in this case, refers to the process of selecting one of the UIFs that were introduced during the instrumentation step and marking it to be excluded from abstraction in all following iterations. Consequently, the number of UIFs that remain in the instrumented programs is reduced by one for every iteration that concludes with a program-level equivalence classification that is neither Eq nor Neq. We are reusing ARDiff's implementation (Badihi et al., 2022) of the abstraction refinement step that applies the three refinement heuristics described in Section 2.3. For a more detailed description of these heuristics, we refer the reader to the ARDiff paper by Badihi et al. (2020).

3.6. Program output

The data that is collected by PASDA is stored in an SQLite database to make it easily available for further processing and analysis. For example, Table 3 shows a subset of the partition-level data that PASDA collects when checking equivalence of `neq_v1` and `neq_v2` from Listing 2. As described in Section 3.2, the collected data contains (a) partition-effects pairs for the two program versions (columns *Path Condition* and *Output v1/v2*), (b) equivalence information (columns *Reachability Classification*, *Output Classification*, and *Overall Classification*), and (c) information about the lines of code that are reached when executing the two target programs with input values that satisfy the corresponding path condition (columns *Covered Lines v1/v2*).

For runs that finish within the configured timeout, PASDA always reports the data collected during its last iteration as the result of its overall analysis. For runs that do *not* finish within the timeout, PASDA either reports the data of (i) the last iteration i_n (which *did* time out) or (ii) the second-to-last iteration i_{n-1} (which did *not* time out) based on the following criteria: if i_n is the only iteration or produces a non-equivalence proof, the data of i_n is reported. Otherwise, the data of i_{n-1} is reported. This special logic is applied to avoid situations where little to no data would be reported when a timeout occurs shortly after a new iteration is started. Note that an equivalence proof can never be produced for timed-out iterations because equivalence can only be proven if all partitions are analyzed.

4. Evaluation

The goals of our evaluation are (i) to compare the equivalence checking accuracy and runtime performance of PASDA to state-of-the-art equivalence checking approaches, and (ii) to measure the accuracy of PASDA's best effort equivalence classification for undecided cases, i.e., program pairs for which no non-/equivalence proof can be found. In line with these goals, we define the following research questions:

- **RQ1**: What is the program-level equivalence classification accuracy of PASDA compared to existing equivalence checking approaches?
- **RQ2**: How accurate are PASDA's best effort equivalence classifications of undecided cases?
- **RQ3**: How many partitions are classified as (maybe) non-/equivalent by PASDA compared to PRV?
- **RQ4**: What is the runtime performance of PASDA compared to existing equivalence checking approaches?

In the following, we first describe the tools included in our evaluation in Section 4.1 and the benchmark cases on which the evaluation was performed in Section 4.2. Throughout Sections 4.3–4.6, we then describe the evaluation results corresponding to the four research questions RQ1–RQ4.

4.1. Evaluated tools/approaches

The equivalence checking approaches that we included in our evaluation are PASDA, ARDiff (Badihi et al., 2020), DSE (Person et al., 2008), and PRV (Böhme et al., 2013). Table 4 shows an overview of the main properties along which these approaches can be differentiated. ARDiff and DSE are both summary-based tools that introduce uninterpreted functions (UIFs) as an abstraction for unchanged parts of the source code. Furthermore, ARDiff iteratively refines these abstractions to mitigate their drawbacks while preserving their benefits at the cost of increased runtimes. PRV, on the other hand, is a partition-based approach that does *not* introduce UIFs into the analyzed programs. Our own approach, PASDA, is also partition-based, but *does* use UIFs and UIF refinement. Additionally, PASDA newly introduces best effort classification of undecided cases, which results in a classification of Unknown results into Maybe_Eq, Maybe_Neq, and Unknown.

ARDiff. The publicly available implementation of ARDiff (Badihi et al., 2022) was modified by us to fix two bugs that caused false positive Neq results in the original ARDiff implementation for a small number of cases. The first fix concerns the incorrect handling of temporary variables introduced by SPF to hold the results of explicit and implicit type casts between integers and doubles. The second fix concerns the incorrect handling of programs that are depth limited for different input partitions. We provide further details about the bugs and fixes in our replication package (Glock et al., 2023).

Table 3
Partition-level data collected by PASDA when checking equivalence of `neq_v1` and `neq_v2` shown in Listing 2.

#	Path condition	Covered lines		Output		Reachability	Output	Overall
		v1	v2	v1	v2	classification	class.	class.
1	$x < 0$	2	8	1	2	Reachable	Neq	Neq
2	$x = 0$	2	8	1	2	Reachable	Neq	Neq
3	$x > 0 \wedge x < 9 \wedge tan(x) < 0 \wedge tan(2x) < 0$	2, 3, 4	8, 9, 10	−1	−1	Maybe_Reach.	Eq	Eq
4	$x > 0 \wedge x < 9 \wedge tan(x) < 0 \wedge tan(2x) > 0$	2, 3, 4	8, 9, 10	−1	0	Maybe_Reach.	Neq	Maybe_Neq
5	$x > 0 \wedge x < 9 \wedge tan(x) > 0 \wedge tan(2x) < 0$	2, 3, 4	8, 9, 10	0	−1	Maybe_Reach.	Neq	Maybe_Neq
6	$x > 0 \wedge x < 9 \wedge tan(x) > 0 \wedge tan(2x) > 0$	2, 3, 4	8, 9, 10	0	0	Maybe_Reach.	Eq	Eq
7	$x > 0 \wedge x = 9 \wedge tan(x) < 0 \wedge tan(2x) < 0$	2, 3, 4	8, 9, 10	0	0	Maybe_Reach.	Eq	Eq
8	$x > 0 \wedge x > 9$	2, 3	8, 9	$tan(x)$	$tan(2x)$	Reachable	Unknown	Unknown

Table 4
Properties of the evaluated equivalence checking approaches.

Tool	Type	UIFs	UIF refinement	Maybe classes
PASDA	Partition-based	✓	✓	✓
ARDiff	Summary-based	✓	✓	✗
DSE	Summary-based	✓	✗	✗
PRV	Partition-based	✗	✗	✗

DSE. For DSE, we also used a fixed version of the tool that is based on a reimplementation by Badihi et al. (2022). Use of the original implementation of DSE is not possible since the tool is not publicly available (Person et al., 2008; Badihi et al., 2020). The reimplementation by Badihi et al. faithfully reconstructs the equivalence checking process described in the DSE paper by Person et al. (2008) but does not include the calculation of behavioral deltas. The fixes that we applied are the same as for ARDiff and are also available in our replication package (Glock et al., 2023).

PRV. No publicly available implementation of PRV exists and we could not get access to the tool by contacting the authors of the PRV paper. We therefore used a variant of PASDA that exhibits the same basic properties as PRV (see Table 4) as a proxy for PRV in our evaluation. Specifically, this variant only performs PASDA's first iteration and does not include best effort classification of undecided cases. While the absolute classification accuracies and runtimes of this variant are likely different from the original PRV implementation, having this variant allows us to compare the relative advantages and disadvantages of PASDA's extensions compared to a PRV-like baseline.

4.2. Benchmark programs

For our evaluation, we used the equivalence checking benchmark built by Badihi et al. (2020) to evaluate ARDiff. The benchmark consists of 141 Java program pairs (73 equivalent, 68 non-equivalent) with seeded changes. All programs are single-threaded and deterministic, and every program is implemented in a single Java class consisting of one or more methods. Program sizes range from 8 lines of code (LOC) to 201 LOC, with an average size of 52.5 LOC. Program constructs used in the benchmark include loops, method calls, and non-linear arithmetic, but do not include recursion. Data types in the programs are limited to integers and doubles. Other data types were not included by the authors of the benchmark because SPF and Z3 only offer limited support for non-primitive data types such as arrays, strings, and other classes.

4.3. RQ1: Equivalence classification accuracy

Setup. We collected the equivalence classification results for all 141 cases in the ARDiff benchmark for each of the four evaluated tools at six different timeout settings: 10 seconds (s), 30 s, 90 s, 300 s, 900 s, and 3600 s. To further improve confidence in the collected results, we conducted five runs for each benchmark:tool:timeout combination, resulting in a total of 16 920 runs across the 3384 five-run-groups. For 3360 of the five-run-groups, all five runs produced the same classification result. For the remaining 24 groups – all of which produced two different results across the five runs in the group – we report the classification result produced by the majority of the runs as the group result. Inter-group classification differences are generally caused by slight runtime differences across the runs of a group, which affect the analysis progress that can be made within the given timeout. Thus, inter-group differences can be observed across all four evaluated tools and all six evaluated timeout settings.

All experiments were run on a 2022 MacBook Air with M2 chip and 24 GB of RAM. Java's `InitialHeapSize` and `MaxHeapSize` settings were left at their default values of 384 MB (i.e., 1/64th of RAM) and 6 GB (i.e., 1/4th of RAM), respectively. The four tools were configured to use a depth limit setting of 10, which we found to offer a good trade-off between tool runtimes and result accuracy in our own testing. Collected results were stored in an SQLite database and analyzed via SQL queries to produce the results of the evaluation. All collected data and the corresponding analysis scripts are available in our replication package (Glock et al., 2023).

Results. Tables 5 and 6 show the program-level classification results of the four tools. Overall, PASDA correctly classifies between 86 of 141 cases (61%) at the 10 s timeout and 104 of 141 cases (74%) at the 3600 s timeout, with correctly classified cases at higher timeouts being (proper) supersets of those that are correctly classified at lower timeouts. Looking at equivalent and non-equivalent cases separately, we find that PASDA correctly classifies 38 of 73 (52%) equivalent and 48 of 68 (61%) non-equivalent cases when using a timeout setting of 10 s. This increases to 51 of 73 (70%) equivalent and 53 of 68 (78%) non-equivalent cases at the 3600 s timeout setting. Most remaining cases are classified as Unknown' (i.e., Maybe_Eq, Maybe_Neq or Unknown — for further details, see Section 4.4). Only a small minority of cases are classified as Depth_Limited, Timeout or Error, with all Error classifications across all tools being caused exclusively by `OutOfMemoryErrors`. No Eq cases are incorrectly classified as Neq or vice versa.

Compared to PRV, PASDA correctly classifies a higher number of Eq cases (e.g., 38 vs. 34 at 10 s and 51 vs. 42 at 3600 s) and the same number of Neq cases. In fact, the set of cases that is correctly classified by PASDA is a (proper) superset of those that are correctly classified by PRV. This is generally guaranteed by our experimental setup (PRV is identical to the first iteration of PASDA). Nevertheless, these results demonstrate that the use of UIFs and UIF refinement can improve the classification accuracy of Eq cases for not only summary-based tools – as demonstrated by ARDiff (Badihi et al., 2020) – but also for partition-based tools such as PASDA, albeit at the cost of longer runtimes (see Section 4.6). Neq cases, on the other hand, generally do not benefit from abstraction via UIFs because it is usually not possible to tell whether non-equivalences that were identified in the abstracted programs (which overapproximate the original program behaviors) can also be observed in the original programs.

Compared to ARDiff, PASDA correctly classifies more cases at all timeout settings (e.g., 86 vs. 78 at 10 s and 104 vs. 86 at 3600 s). For Eq cases, PASDA correctly classifies fewer cases than ARDiff at low timeouts (e.g., 38 vs. 45 at 10 s) and more at high timeouts (e.g., 51

Table 5
Program-level Eq, Neq, and Unknown' classifications of the four evaluated tools at the six evaluated timeout settings. To aid comparison across tools, Unknown' shows an aggregate of Maybe_Eq, Maybe_Neq, and Unknown classifications for PASDA.

Tool	Expected	Eq						Neq						Unknown'					
		10 s	30 s	90 s	300 s	900 s	3600 s	10 s	30 s	90 s	300 s	900 s	3600 s	10 s	30 s	90 s	300 s	900 s	3600 s
PASDA	Eq	38	41	45	47	50	51	0	0	0	0	0	0	32	31	27	25	23	22
PASDA	Neq	0	0	0	0	0	0	48	50	50	52	52	53	17	17	17	15	16	14
ARDiff	Eq	45	48	48	48	48	49	0	0	0	0	0	0	16	16	18	19	19	18
ARDiff	Neq	0	0	0	0	0	0	33	35	35	36	37	37	27	26	26	24	22	21
DSE	Eq	30	30	30	30	30	30	0	0	0	0	0	0	32	36	38	38	37	37
DSE	Neq	0	0	0	0	0	0	21	21	21	21	21	21	39	40	40	40	40	41
PRV	Eq	34	37	39	41	41	42	0	0	0	0	0	0	36	35	33	31	32	31
PRV	Neq	0	0	0	0	0	0	48	50	50	52	52	53	17	17	17	15	16	15

Table 6
Program-level Depth_Limited, Timeout, and Error classifications of the four evaluated tools at the six evaluated timeout settings.

Tool	Expected	Depth_Limited						Timeout						Error					
		10 s	30 s	90 s	300 s	900 s	3600 s	10 s	30 s	90 s	300 s	900 s	3600 s	10 s	30 s	90 s	300 s	900 s	3600 s
PASDA	Eq	1	0	0	1	0	0	2	1	1	0	0	0	0	0	0	0	0	0
PASDA	Neq	1	0	0	1	0	0	2	1	1	0	0	0	0	0	0	0	0	1
ARDiff	Eq	0	0	0	0	0	0	12	9	7	6	4	4	0	0	0	0	2	2
ARDiff	Neq	0	0	0	0	0	0	8	7	7	7	6	5	0	0	0	1	3	5
DSE	Eq	0	0	0	0	0	0	11	7	5	5	5	4	0	0	0	0	1	2
DSE	Neq	0	0	0	0	0	0	8	7	7	7	6	5	0	0	0	0	1	1
PRV	Eq	1	0	0	1	0	0	2	1	1	0	0	0	0	0	0	0	0	0
PRV	Neq	1	0	0	1	0	0	2	1	1	0	0	0	0	0	0	0	0	0

vs. 49 at 3600 s). For Neq cases, PASDA correctly classifies more cases than ARDiff at all timeout settings (e.g., 48 vs. 33 at 10 s and 53 vs. 37 at 3600 s). While there is significant overlap across the sets of cases that are correctly classified by PASDA and ARDiff, neither is a superset of the other. For example, at the 300 s timeout, 44 of the 73 Eq cases are correctly classified by both tools, whereas 3 are only correctly classified by PASDA and 4 are only correctly classified by ARDiff. At the same 300 s timeout, 35 of the 68 Neq cases are correctly classified by both tools, 17 only by PASDA, and 1 only by ARDiff. PASDA generally performs better than ARDiff on cases that are best analyzed without abstraction, i.e., without using UIFs to represent unchanged parts of the source code. This is because PASDA checks equivalence without abstraction in its first iteration, whereas ARDiff does this in its last iteration (which generally is not reached within the given timeout). On the other hand, PASDA generally performs worse than ARDiff for Eq cases that ARDiff *can* fully analyze within the given timeout but PASDA *cannot* due to its higher runtime requirements (see Section 4.6 for further details about runtime differences).

Compared to DSE, which is identical to ARDiff's first iteration, PASDA correctly classifies more Eq cases as well as more Neq cases at all timeout settings. Intuitively, the comparatively low classification accuracy of DSE can be attributed to the fact that DSE introduces UIFs not only to replace complex program constructs that are difficult to reason about (e.g., loops, non-linear arithmetic, etc.) but also for simple ones (e.g., variable assignments, linear arithmetic, etc.). This results in unnecessary overapproximations that hinder non-/equivalence proofs. PASDA and ARDiff mitigate these negative effects through their abstraction refinement process as proposed by Badihi et al. (2020). However, this comes at the cost of longer runtimes (see Section 4.6) because multiple analysis iterations have to be performed to find an appropriate level of abstraction.

Answer to RQ1: PASDA achieves an overall program-level equivalence classification accuracy between 61% at the 10 s timeout setting and 74% at the 3600 s timeout setting. Compared to the three existing tools included in our analysis, which achieve classification accuracies between 36–58% at the 10 s timeout setting and 36–67% at the 3600 s timeout setting, PASDA therefore correctly classifies a larger number of cases at all six of the analyzed timeout settings.

4.4. RQ2: Best effort classification accuracy

Setup. To evaluate the accuracy of PASDA's best effort equivalence classifications, we used the same basic setup as for RQ1. For the purpose of accuracy calculations, we consider Maybe_Eq classifications as correct if they are reported for Eq benchmark cases, and as incorrect if they are reported for Neq benchmark cases. Similarly, we consider Maybe_Neq classifications as correct when reported for Neq cases and as incorrect when reported for Eq cases. To allow us to compare the effectiveness of PASDA's best effort classification approach across different baselines, we retrofitted it to the three existing tools ARDiff, DSE, and PRV. This causes a classification of their Unknown results into Maybe_Eq, Maybe_Neq and Unknown. None of their other classifications are affected by this change.

Results. Table 7 shows the best effort classification results for all benchmark cases that were previously classified as Unknown (see columns Unknown' in Table 6). For PASDA, around 10%–15% of these cases retain their Unknown classifications (e.g., 5 of 49 at the 10 s timeout and 4 of 36 at 3600 s), signifying that none of the fully analyzed partitions provide any indication of potential non-/equivalence. Around 65%–80% of cases (e.g., 38 of 49 at 10 s and 27 of 36 at 3600 s) are classified as Maybe_Eq and 10%–25% are classified as Maybe_Neq (e.g., 6 of 49 at 10 s and 5 of 36 at 3600 s).

Among the Maybe_Eq cases, around 70%–75% are correctly classified (e.g., 28 of 38 at 10 s and 19 of 27 at 3600 s), whereas the remaining 25%–30% are incorrectly classified. Incorrect Maybe_Eq classifications arise for the following two reasons: (i) the programs were only partially analyzed (due to the configured timeout and/or depth limit) and the non-equivalent partitions were *not* reached by the analysis, (ii) the non-equivalent partitions *were* reached by the analysis but non-equivalence could not be proven due to limitations of Z3, which we use for non-/equivalence checks. These reasons directly follow from the definitions of our classification heuristics (see Sections 3.3 and 3.4) and could thus potentially be improved upon through the use of a different set of heuristics. We leave such refinements for future research.

Among the Maybe_Neq cases, fluctuations in classification accuracy are higher than for Maybe_Eq cases, with correct classifications reaching 55%–85% across the different timeouts (e.g., 4 of 6 at 10 s and 4 of 5 at 3600 s). Incorrect Maybe_Neq classifications primarily arise

Table 7
Program-level Maybe_Eq, Maybe_Neq, and Unknown classifications at the six evaluated timeout settings. ARDiff, DSE, and PRV were modified by us to differentiate between these three classes for cases that were previously classified as Unknown.

Tool	Expected	Maybe_Eq						Maybe_Neq						Unknown					
		10 s	30 s	90 s	300 s	900 s	3600 s	10 s	30 s	90 s	300 s	900 s	3600 s	10 s	30 s	90 s	300 s	900 s	3600 s
PASDA	Eq	28	28	23	19	20	19	2	2	3	4	1	1	2	1	1	2	2	2
PASDA	Neq	10	9	8	7	7	8	4	4	5	5	5	4	3	4	4	3	4	2
ARDiff	Eq	8	9	10	11	11	11	5	1	1	1	0	0	3	6	7	7	8	7
ARDiff	Neq	2	2	2	2	2	2	16	10	10	8	7	6	9	14	14	14	13	13
DSE	Eq	5	6	6	6	6	6	19	20	21	21	21	21	8	10	11	11	10	10
DSE	Neq	2	2	2	2	2	2	32	33	33	33	34	35	5	5	5	5	4	4
PRV	Eq	33	32	30	28	29	28	0	0	0	0	0	0	3	3	3	3	3	3
PRV	Neq	11	10	9	8	8	8	2	2	3	3	3	3	4	5	5	4	5	4

Table 8
Occurrence frequencies of different partition-level classifications when running PASDA with a timeout setting of 300 s. Partition-level classifications (columns 4–9) that cannot occur for the corresponding program-level classification (column 3) are marked with a "–".

#	Program-level expected	Program-level actual	Average % of ⟨Classification⟩ partitions per run: Eq	Neq	Maybe_Eq	Maybe_Neq	Unknown	Depth_Limited
1	Eq	Eq	100.0	–	–	–	–	–
2	Eq	Maybe_Eq	67.3	–	0.0	–	0.9	31.8
3	Eq	Maybe_Neq	54.8	–	1.9	21.9	1.9	19.6
4	Eq	Unknown	–	–	–	–	100.0	0.0
5	Eq	Depth_Limited	–	–	–	–	–	100.0
6	Neq	Neq	22.3	42.5	0.0	0.9	7.3	27.0
7	Neq	Maybe_Eq	58.4	–	0.0	–	35.1	6.6
8	Neq	Maybe_Neq	35.9	–	0.0	33.7	1.5	28.9
9	Neq	Unknown	–	–	–	–	60.0	40.0
10	Neq	Depth_Limited	–	–	–	–	–	100.0
11	*	*	50.8	20.5	0.1	3.5	6.1	18.9

when (i) the observed non-equivalent program behaviors only manifest due to the overapproximations caused by the introduction of UIFs or (ii) partitions with a reachability classification of Maybe_Reachable and an output classification of Neq are actually Unreachable but could not be proven to be so due to limitations of Z3, which we use for reachability checks. Again, the classification accuracy that is achieved for such cases could potentially be improved upon through the use of a different set of classification heuristics. Alternatively, the concrete input values that are provided by PASDA to demonstrate non-equivalence could be used to check whether concrete executions of the two original program versions produce non-equivalent results for these inputs. If they do, the result is a true positive. Otherwise, it is a false positive. Note that this approach *cannot* be used to check the correctness of Maybe_Eq classifications. This is because, in the Maybe_Eq case, we need to prove that the two programs are equivalent for *all* inputs, which cannot be achieved by checking a single set of input values.

Best effort classification accuracies show similar trends for ARDiff, DSE, and PRV as they do for PASDA. Specifically, correct classifications are consistently above 60% for both Maybe_Eq and Maybe_Neq cases at all timeout settings. For high timeout settings of ARDiff and all timeout settings of PRV, classification accuracies of Maybe_Neq cases even reach 100%. However, this is not guaranteed in general, since PRV can still produce false positive Maybe_Neq classifications in the presence of incorrect Maybe_Reachable reachability classifications, and ARDiff can additionally produce false positive Maybe_Neq classifications due to overapproximations introduced by UIFs. ARDiff and DSE produce more Maybe_Neq results than PASDA and PRV because the two partition-based tools can often prove the corresponding cases to be Neq instead. Similarly, PASDA produces fewer Maybe_Eq results than PRV because it can often prove them to be Eq instead.

As described in Section 3.4, *program-level* (best effort) classifications are generally derived from *partition-level* (best effort) classifications through a set of classification heuristics. Exceptions to this rule are (i) runs during which an error occurs, which are always classified as Error, and (ii) runs for which not even a single partition can be analyzed within the given timeout, which are always classified as Timeout. For the retrofitted variants of ARDiff and DSE, we consider the single program-level equivalence check to be representative of a partition that covers the whole input space. Otherwise, the same classification heuristics are applied to produce (best effort) classification results.

To provide a better understanding of these heuristics, Table 8 shows how often each partition-level result is observed, on average, across runs with a particular program-level classification when running PASDA with a timeout setting of 300 s. By definition, a program-level result of Eq is only reported when PASDA is able to analyze all partitions within the given timeout and proves them all to be Eq (see row #1). Maybe_Eq is reported when some partitions are classified as Eq or Maybe_Eq, but none are classified as Neq or Maybe_Neq (see rows #2 and #7). Otherwise, cases are classified based on the partition with the highest priority, where Neq > Maybe_Neq > Unknown > Depth_Limited. Thus, program-level Neq cases are the only ones that can contain Neq partitions, but might also contain all other types of partitions (see row #6). Similarly, Neq and Maybe_Neq cases are the only ones that can contain Maybe_Neq partitions (see rows #3, #6, and #8). Depth_Limited cases, on the other hand, can only contain Depth_Limited partitions (see rows #5 and #10), since any other partition-level result would force a different program-level classification to be chosen. For the same reason, Unknown cases can only contain Unknown and Depth_Limited partitions (see rows #4 and #9).

Answer to RQ2: Through the introduction of our best effort classification approach, 65-80% of previously Unknown cases are classified as Maybe_Eq, 10–25% as Maybe_Neq, and 10–15% retain their Unknown classifications. Maybe_Eqs are correct for 70–75% of cases across the six evaluated timeout settings between 10 s and 3600 s. Maybe_Neqs are correct for 55–85% of cases. We observed similar best effort classification accuracies when retrofitting PASDA's best effort classification approach to ARDiff, DSE and PRV.

4.5. RQ3: Partition-level non-/equivalence proofs

Setup. To answer RQ3, we executed PASDA and PRV five times each for every case in the benchmark across each of the six timeout settings

Table 9
Overall partition-level equivalence classification results of PASDA and PRV across the six analyzed timeout settings.

Tool	Timeout (s)	# Partitions	Average % of ⟨Classification⟩ Partitions					
			Eq	Neq	Maybe_Eq	Maybe_Neq	Unknown	Depth_Limited
PASDA	10	3298	52.5	23.1	0.0	2.8	5.3	16.3
PASDA	30	4724	47.8	20.7	0.1	2.1	6.0	23.2
PASDA	90	5155	52.6	20.0	0.1	3.2	6.1	18.0
PASDA	300	5502	50.8	20.5	0.1	3.5	6.1	18.9
PASDA	900	5430	52.3	20.9	0.1	1.3	6.1	19.3
PASDA	3600	5535	51.4	20.4	0.1	1.5	5.7	20.9
PRV	10	3500	51.4	21.8	–	–	7.8	19.0
PRV	30	4804	47.8	20.4	–	–	6.9	24.9
PRV	90	5361	49.6	19.2	–	–	6.7	24.5
PRV	300	5727	47.8	19.7	–	–	6.5	26.1
PRV	900	5768	47.7	19.6	–	–	6.4	26.3
PRV	3600	5809	47.4	19.7	–	–	6.4	26.5

of 10 s, 30 s, 90 s, 300 s, 900 s, and 3600 s. ARDiff and DSE are not included in the partition-level analysis since neither tool produces partition-level results. The remaining hardware and tool configurations are the same as for RQ1.

Results. Table 9 provides an overview of the partition-level equivalence classification results produced by PASDA and PRV across the six analyzed timeout settings. In total, PASDA identifies 3298 partitions across the 705 runs (i.e., 5 runs for each of the 141 benchmark cases) at the 10 s timeout setting. This number increases to 5535 partitions at the 3600 s timeout setting, whereas PRV identifies between 3500 partitions at 10 s and 5809 at 3600 s. The number of reported partitions is generally lower for PASDA than for PRV because of PASDA's iterative abstraction and refinement process. After all, partition counts can decrease if the data of the last (partially analyzed) iteration is reported as PASDA's analysis result (see Section 3.6), and partition counts can further fluctuate as UIFs are added and removed throughout PASDA's analysis process. This also explains why PASDA reports fewer partitions when raising its timeout setting from 300 s to 900 s.

Partition counts across different benchmark cases are highly skewed. For example, partition counts reported by PASDA at the 300 s timeout setting have a skewness value of 1.2 (mean: 7.8, median: 4, mode: 3), which indicates that most cases in the benchmark have relatively few partitions, but some cases exist that have much higher partition counts. High partition counts are most commonly found among cases that contain (nested) loops and/or many (nested) if constructs. Furthermore, Neq cases tend to have higher partition counts than corresponding Eq cases. Intuitively, this is because the introduction of semantic changes often causes some Eq partitions to be split into multiple Eq and Neq subpartitions.

Looking at PASDA's classification results, we find that it proves 50.8% of identified partitions to be Eq and 20.5% to be Neq at the 300 s timeout setting. For the remaining 28.7% of Maybe_Eq (0.1%), Maybe_Neq (3.5%), Unknown (6.1%), and Depth_Limited (18.9%) partitions, no non-/equivalence proofs can be provided due to (i) depth limiting, (ii) UIF-induced overapproximations, and (iii) limitations of Z3, which we use for reachability and non-/equivalence checks. As timeouts are changed, the proportions held by the different classifications fluctuate by a few percentage points, but no clear upward or downward trend emerges for any of them. However, since a larger number of partitions is analyzed at higher timeouts, this means that a larger part of the overall input space can be proven to be Eq or Neq, or classified as either Maybe_Eq or Maybe_Neq. For example, only 452 of 705 runs (64%) finish at least one analysis iteration (i.e., fully analyze the original programs without UIFs) at the 10 s timeout, whereas 620 (88%) runs finish at least one iteration at 3600 s.

Compared to PRV, PASDA provides (best effort) non-/equivalence classifications for a larger percentage of identified partitions across all six analyzed timeout settings. More specifically, PASDA classifies 0.0–4.6% more partitions as Eq, 0.3–1.3% more as Neq, 0.0–0.1% as Maybe_Eq, and 1.5–3.2% as Maybe_Neq. Consequently, the percentage of partitions for which no (best effort) non-/equivalence classifications can be provided by PASDA is consistently lower than that of PRV, reaching a 0.4–2.5% lower percentage of Unknown partitions and a 1.7–7.2% lower percentage of Depth_Limited partitions. These results match our expectations. After all, the introduction of UIFs intends to enable Eq proofs for previously Unknown and Depth_Limited partitions, but can also lead to an increase of other non-Depth_Limited classifications if no Eq proof can be provided. Additionally, the introduction of best effort classifications intends to replace some previously Unknown results with Maybe_Eq or Maybe_Neq.

Answer to RQ3: Across the six analyzed timeout settings, PASDA provides non-/equivalence *proofs* for 70.7–78.4% of identified partitions, whereas PRV provides such proofs for 67.1–73.2% of identified partitions. At identical timeout settings, PASDA therefore provides non-/equivalence proofs for 2.5–7.4% more of the total partitions than PRV. Additionally, PASDA provides *best effort* non-/equivalence classifications for 1.4–3.6% of identified partitions, whereas PRV, by design, cannot provide any best effort classifications.

4.6. RQ4: Runtime performance

Setup. To answer RQ4, we reused the basic setup from RQ1 described before. Thus, each of the four tools was executed five times for each benchmark case at each of the six analyzed timeout settings from 10 s to 3600 s. Runtime measurements were taken for each processing step of the four tools using a custom wrapper around the Apache Commons `StopWatch` class (The Apache Software Foundation, 2007). On average, there is around a 10% difference between the fastest and the slowest run of each five-run-group. We present the runtime results based on the median runtimes of each group to mitigate the influence of occasional random outliers. The full runtime data is available in our replication package (Glock et al., 2023).

Results. Fig. 2 shows the runtime distributions of the four analyzed tools at timeout settings of 10 s, 300 s, and 3600 s. Note that the measured runtimes are highly skewed for all tools, reaching skewness values between 0.91 and 2.00 across the different tool:timeout combinations. Consequently, median runtimes remain largely unchanged across the different timeouts, ranging from 2.2 s to 4.8 s across all tools. Mean runtimes, on the other hand, increase significantly as timeouts are raised. More specifically, they start at around 5 s for all tools at the 10 s timeout setting and go up to 300–700 s across the different tools at the 3600 s setting.

For Eq cases, DSE generally has the shortest runtimes of all tools because it only performs a single analysis iteration (whereas PASDA and ARDiff perform multiple iterations) and often abstracts program constructs such as loops that would otherwise be expensive to analyze (whereas PRV does not use any abstraction). For Neq cases, PRV generally has the shortest runtimes because it only performs a single iteration

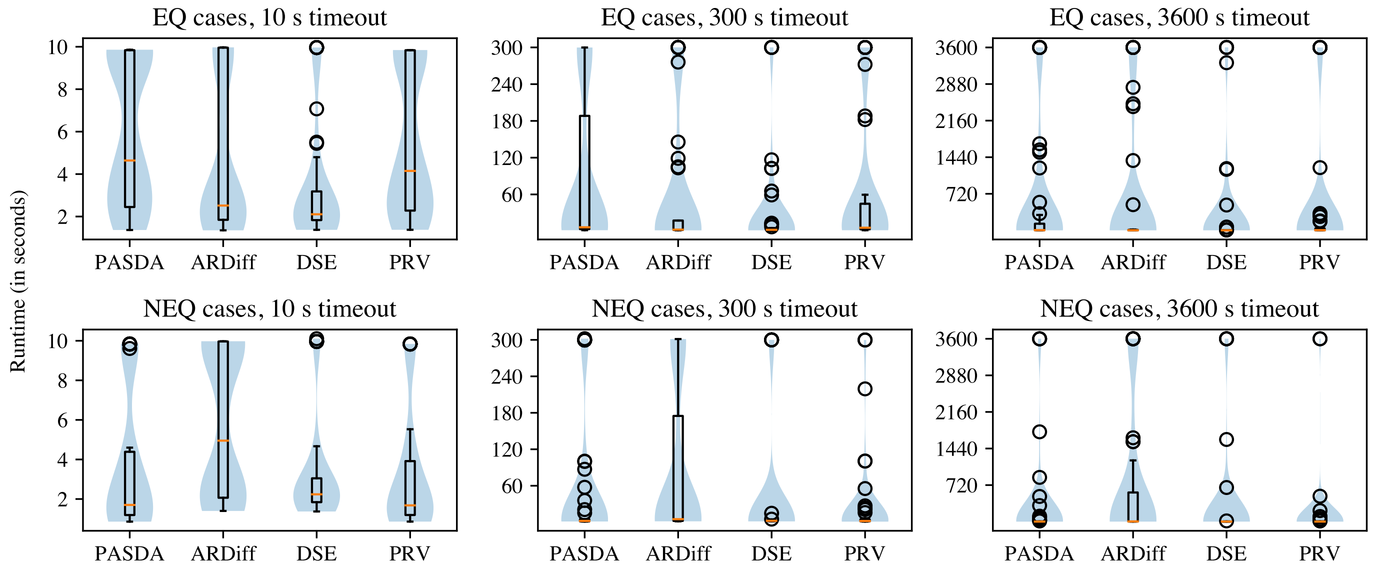


Fig. 2. Runtime distributions of the four evaluated tools (PASDA, ARDiff, DSE, PRV) at timeout settings of 10 s, 300 s, and 3600 s per type of benchmark case (Eq vs. Neq).

(whereas PASDA performs multiple iterations) and can provide non-equivalence proofs without fully analyzing the two target programs (whereas ARDiff and DSE have to fully analyze the target programs). PASDA has the *longest* runtimes for Eq cases and ARDiff for Neq cases.

On average, PASDA takes around 6%–17% longer to analyze the same cases than PRV. This holds for both Eq as well as Neq benchmark cases at all timeout settings, though the difference in runtimes is smallest at low timeouts and largest at high timeouts. Since PRV is identical to PASDA's first iteration in our setup, runtime differences can be attributed to the iterative abstraction refinement process used by PASDA. At the 10 s timeout setting, PASDA performs more than one iteration for 18% of Eq and 9% of Neq cases. These numbers go up as timeouts are increased because fewer cases are interrupted by the timeout before the analysis reaches later iterations, which also explains the larger runtime difference between PRV and PASDA at higher timeouts. At the 3600 s timeout setting, PASDA performs more than one iteration for 32% of Eq cases and 19% of Neq cases. The number of cases for which PASDA runs into the timeout decreases from 42% of cases at 10 s to 18% at 3600 s.

Compared to ARDiff, PASDA is generally slower to provide equivalence proofs but faster to provide non-equivalence proofs. More specifically, for Eq cases that both tools classify correctly, PASDA takes, on average, 20%–35% longer to produce equivalence proofs than ARDiff across the different timeouts. The primary reason for this difference is that ARDiff only executes all paths in each of the two target programs once ($O(m + n)$). PASDA, on the other hand, executes every path in the second program once for every path in the first program ($O(m \cdot n)$) when symbolically executing the product program described in Listing 3. However, many paths are only partially explored because they are quickly identified to be unreachable, which significantly reduces the runtime impact caused by PASDA's higher runtime complexity. For Neq cases, ARDiff takes around 2–3 times as long to provide non-equivalence proofs as PASDA does. This is mainly because ARDiff only checks non-/equivalence after both programs have been fully explored by the symbolic execution. PASDA, on the other hand, checks non-/equivalence multiple times throughout its execution, i.e., every time a single program path has been fully explored. Consequently, PASDA can often provide a non-equivalence proof after having analyzed only a small subset of all program paths.

Table 10 shows the average runtimes of PASDA's processing steps across the six analyzed timeout settings from 10 s to 3600 s. The largest factor contributing to PASDA's overall runtime is the symbolic execution step, which takes an average of 4.5 s at the 10 s timeout setting and increases to 454.3 s at the 3600 s timeout setting. The partition classification and abstraction refinement steps also show noticeable runtime increases as timeouts are raised, reaching average runtimes of 115.4 s and 31.4 s, respectively, at the 3600 s timeout. Runtime requirements for the remaining steps of PASDA's analysis process (i.e., initialization, instrumentation, program classification, and finalization) remain largely the same at around 1 s or less per step across the different timeout settings. As such, they only affect PASDA's overall runtimes in relatively minor ways, particularly at higher timeouts where the runtime requirements of symbolic execution, partition classification, and abstraction refinement (all three of which involve the use of constraint solving) are much larger.

Table 10
Average runtimes of PASDAs processing steps at the six evaluated timeout settings.

Step	Ø Runtime (s)					
	10 s	30 s	90 s	300 s	900 s	3600 s
Initialization	1.1	1.0	1.1	1.2	1.1	1.2
Instrumentation	0.2	0.3	0.3	0.3	0.3	0.4
Symb. Execution	4.5	11.4	26.4	67.1	147.3	454.3
Part. Classification	0.3	1.1	4.0	14.0	36.1	115.4
Prog. Classification	0.0	0.0	0.0	0.0	0.0	0.0
Abstr. Refinement	0.0	0.1	0.6	2.5	10.1	31.4
Finalization	0.0	0.0	0.0	0.0	0.0	0.0

Answer to RQ4: For Eq cases, ARDiff, DSE, and PRV generally have 10–50% shorter runtimes than PASDA. For Neq cases, PRV's runtimes are generally 5–15% shorter than PASDA's whereas the runtimes of ARDiff and DSE are often multiple times as long. The processing step that makes up the largest portion of PASDA's overall runtimes is the symbolic execution step, followed by the partition classification step and the abstraction refinement step.

5. Discussion

In the following subsections, we discuss the benefits that PASDA provides compared to existing equivalence checking approaches, outline potential use cases for PASDA's best effort equivalence classifications, and list possible threats to the validity of our results as well as how we addressed them.

5.1. Benefits of PASDA

As shown by our evaluation, there are two main dimensions along which PASDA achieves measurable improvements compared the current state-of-the-art for equivalence checking of software programs: (i) PASDA provides non-/equivalence *proofs* for a larger percentage of analyzed programs and partitions than existing tools (see RQ1 and RQ3), and (ii) PASDA provides *best effort* equivalence classifications of Maybe_Eq and Maybe_Neq for some programs and partitions that existing tools simply classify as Unknown (see RQ2 and RQ3). PASDA achieves these improvements at the cost of runtimes that are moderately longer than those of existing tools (see RQ4). Increases in equivalence checking accuracy directly benefit use cases that depend on equivalence checking as part of their analysis process. Such use cases are quite diverse, including the verification of compiler optimizations (Dahiya and Bansal, 2017), refactoring assurance for developers (Person et al., 2008), test suite amplification (Danglot et al., 2019), and many others beyond those (see, e.g., Sun et al. (2016), Mora et al. (2018) and Mercaldo and Santone (2021)).

Regarding the potential benefits of best effort equivalence classifications, it is important to note that these classifications always arise as the result of a partial proof of non-/equivalence. After all, a program-level result of Maybe_Eq is reported when PASDA proves that the two target programs produce equivalent outputs for *some* partitions but cannot provide a full equivalence proof because (i) some outputs cannot be proven to be non-/equivalent (i.e., are Unknown) or (ii) some partitions are not reached by PASDA's analysis due to the used depth limit or timeout settings. Similarly, a result of Maybe_Neq is reported when PASDA *cannot* prove whether a specific path through the two target programs is reachable by a concrete execution of the product program but *can* prove that some outputs are non-equivalent *iff* the path *is* reachable (or vice versa). As such, the introduction of best effort equivalence classifications leads to a more complete description of the overall program behavior than would be achieved without them.

5.2. Potential use cases for best effort classifications

Based on the understanding that best effort equivalence classifications are always supported by a *partial* non-/equivalence proof, we envision the potential use cases for this information to include, for example, test case prioritization (Khatibsyarbini et al., 2018), fault localization (Wong et al., 2016), and general support for manual debugging tasks. What these use cases have in common is that they all benefit from more complete descriptions of program behaviors even in the absence of full non-/equivalence proofs. In the following, we discuss how the information provided by PASDA could be integrated into corresponding approaches to improve the results that they produce. A more in-depth investigation of the derived benefits is beyond the scope of this paper and therefore left for future work.

Test case prioritization. In test case prioritization, a commonly targeted goal is to execute tests that are likely to reveal regressions as early as possible in the execution of a given test suite (Khatibsyarbini et al., 2018). Partition-level equivalence checking results can be used to achieve this goal by prioritizing tests whose input values fall within input partitions that are classified as Neq over those that are classified as Unknown and, lastly, Eq. Best effort classifications enable a better informed ranking approach that additionally prioritizes Maybe_Neq partitions over Unknown and Maybe_Eq ones, resulting in a prioritization of Neq > Maybe_Neq > Unknown > Maybe_Eq > Eq. For example, assuming a test suite for the non-equivalent programs in Listing 2 exists, test cases that use inputs that match path conditions #1 or #2 (Neq) in Table 3 should be executed first. Next, test cases matching partitions #4 or #5 (Maybe_Neq) should be executed, followed by test cases matching partition #8 (Unknown), and finally #3, #6, or #7 (Eq).

Fault localization. The goal of fault localization approaches is to identify parts of the source code that are likely to be the root cause of observed program crashes or test failures (Wong et al., 2016). Commonly, this is achieved through spectrum-based fault localization approaches that assign a suspiciousness value to each line of code based on the number of times the line is executed in passing and failing tests (Idrees and Beszédes, 2022). Partition-level equivalence checking results can be used instead of test outcomes by using the number of times a line is covered by Eq and Neq partitions to replace successful and failing test execution counts. For example, we can use $s = N_{cf} + N_{us}$ (Wong et al., 2016) as our test outcome-based suspiciousness formula. Here, N_{cf} is the number of failing tests that cover a line, and N_{us} is the number of successful tests that do *not* cover the same line. This formula can be transformed to $s = N_{cNEQ} + N_{uEQ}$, where N_{cNEQ} and N_{uEQ} are the number of Neq and Eq partitions a line is (not) covered by. To integrate best effort classifications into this formula, we could further transform it to $s = N_{cNEQ} + w_1 \cdot N_{cMAYBE_NEQ} + w_2 \cdot N_{uMAYBE_EQ} + N_{uEQ}$. Here, N_{cMAYBE_NEQ} and N_{uMAYBE_EQ} are the number of Maybe_Neq and Maybe_Eq partitions a line is (not) covered by. These are modified by two weighting factors w_1 and w_2 to account for the uncertainty associated with best effort classification results.

Manual debugging. To aid manual debugging processes, PASDA outputs various partition-level information alongside its program-level equivalence checking classifications (see Section 3.6). Such information has been found to benefit developers' understanding of program behaviors and program analysis results by multiple existing studies with developers (LaToza and Myers, 2010; Parnin and Orso, 2011; Winter et al., 2022). While the reported information is most comprehensive if non-/equivalence can be proven, PASDA still provides parts of it even in the absence of an overall non-/equivalence proof. For example, a program-level classification of Maybe_Eq is always accompanied by partition-level equivalence proofs for some (but not all) of a program's input space. Similarly, a partition-level classification of Maybe_Neq always contains a proof of either reachability or output non-equivalence (but not both). As a practical example of these properties, we can observe that PASDA classifies the two equivalent programs in Listing 1 as Maybe_Eq. To support this classification, PASDA provides partition-level equivalence proofs as well as line coverage information for all input values up to a pre-configured depth limit for loop exploration. In contrast, ARDiff (Badihi et al., 2020) classifies these programs as Unknown and does not provide any further information beyond this. Similarly, given the two non-equivalent programs in Listing 2, PASDA classifies partitions #4 and #5 shown in Table 3 as Maybe_Neq and provides output non-equivalence proofs as well as line coverage information for both of them. PRV (Böhme et al., 2013), on the other hand, classifies these partitions as Unknown and does not provide any additional information for them.

5.3. Threats to validity

Construct validity. Benchmarking is widely established as one of the main methodologies for comparing the runtime performance and accuracy of newly developed approaches to the state-of-the-art. We used a well established benchmark (Badihi et al., 2020) that was also used by ARDiff to measure the accuracy and runtime performance of our approach, thus demonstrating our approach's effectiveness and efficiency in an evidence-based and reproducible way.

Internal validity. To mitigate threats to the internal validity of our results, we manually validated the outputs (program-level and partition-level equivalence classifications, partition-effects pairs, coverage information, etc.) produced by the four tools for a random sample of investigated cases. During this validation, we did not identify any outputs that we would deem incorrect. The results of this manual validation are further supported by the results of our evaluation, which demonstrate that neither of the analyzed tools produced false positive Eq or Neq classifications for any of the analyzed cases.

External validity. Our findings might not generalize to cases beyond those that we investigated in our evaluation. To mitigate this threat, we conducted our evaluation based on the existing ARDiff benchmark (Badihi et al., 2020), which was itself constructed by combining and extending multiple smaller benchmarks for symbolic execution-based equivalence checking. Still, certain program constructs such as recursion, strings, and arrays, which are difficult for symbolic execution and automated decision procedures to reason about, are missing from the benchmark, which limits the generalizability of our results. In particular, our results are unlikely to generalize to (larger) real-world cases due to the inherent limitations of the underlying decision procedures (see Section 2.1), though we share this limitation with all existing equivalence checking approaches.

6. Related work

As described throughout this paper, we reuse several ideas that were originally proposed by the authors of DSE (Person et al., 2008), ARDiff (Badihi et al., 2020), and PRV (Böhme et al., 2013) in our approach. For example, like DSE, we also replace unchanged parts of the source code with UIFs. Furthermore, like ARDiff, we iteratively refine the set of introduced UIFs to improve overall equivalence checking accuracy. Finally, like PRV, we conduct a partition-based analysis of the two target programs. Additionally, PASDA provides best effort equivalence classifications for cases that it cannot prove to be either equivalent or non-equivalent, whereas existing approaches such as UC-KLEE (Ramos and Engler, 2011), PEQcheck (Jakobs, 2021), and many more that have been developed throughout the years (e.g., Godlin and Strichman (2009), Backes et al. (2013), Beyer et al. (2013), Felsing et al. (2014) and Fedyukovich et al. (2016)) classify such cases as Unknown and do not provide any further information beyond this.

To further improve the utility of its outputs, PASDA provides supporting information alongside its (best effort) equivalence classification results. This information includes partition-level equivalence checking results as well as execution traces and concrete and symbolic inputs and outputs. Similar information is provided by some existing equivalence checking approaches. For example, PRV (Böhme et al., 2013) also provides partition-level equivalence checking results. DSE (Person et al., 2008) and DiSE (Person et al., 2011; Yang et al., 2014) both provide behavioral deltas for non-equivalent partitions. Furthermore, Mercer et al. (2012) present an extension for Java PathFinder (Havelund and Pressburger, 2000; The Java PathFinder Contributors, 2005) that visualizes program statements that DiSE identifies to be impacted by identified changes via highlighting of source code and control flow graphs. Symdiff (Lahiri et al., 2012) provides a set of concrete inputs that demonstrates non-equivalence, and highlights the corresponding execution paths in the source code. Partush and Yahav (2014) use abstract interpretation (Cousot and Cousot, 1977) to characterize changed as well as unchanged program behavior.

Various other approaches exist that offer different representations of software changes but do not provide functional non-/equivalence proofs. For example, Le and Pattison (2014) propose multiversion interprocedural control flow graphs to represent changes in control flow across any number of program versions. LSdiff offers mechanisms that can group related changes and describe identified inconsistencies within these groups (Kim and Notkin, 2009). Dex applies graph differencing to semantic graphs of programs and calculates summary statistics from the differencing results (Raghavan et al., 2004). Furthermore, software evolution management tools such as Diffbase (Wu et al., 2021b) and EvoMe (Wu et al., 2021a) provide platforms for processing, storage, and exchange of change information about software programs across different analyses, and offer search features to make access to this data more convenient and less time consuming (Di Grazia et al., 2022). Integrating PASDA's change information into such a tool might be a fruitful direction for future work.

Additional improvements of PASDA's accuracy and runtime performance could potentially be achieved through the integration of optimizations proposed by other equivalence checking, regression verification, and symbolic execution approaches. For example, improved path selection techniques such as the ones proposed by directed incremental symbolic execution (Yang et al., 2014) and shadow symbolic execution (Kuchta et al., 2018) could be used to prioritize paths that exercise changed parts of the source code. Summarization techniques for loops (Godefroid and Luchaup, 2011) and other program regions (Sharma et al., 2020) could be added to alleviate the path explosion problem. Furthermore, PASDA's analysis algorithm could be adapted to rely not only on symbolic execution, but to also include other analysis techniques such as fuzzing, a combination with which HyDiff (Noller et al., 2020) achieved promising results. However, such optimizations are largely orthogonal to PASDA's best effort equivalence classifications. After all, equivalence checking is undecidable in general (Godlin and Strichman, 2009). Thus, there will always be programs and partitions for which no overall non-/equivalence proof can be provided and which, therefore, might benefit from best effort equivalence classifications and corresponding partial non-/equivalence proofs as reported by PASDA.

7. Conclusions

We presented PASDA, our partition-based semantic differencing approach with best effort classification of undecided cases. PASDA conducts multiple analysis iterations in which it aggregates partition-level equivalence checking results obtained by symbolically executing a product program created from two target programs to produce program-level equivalence classifications. If neither equivalence nor non-equivalence can be formally proven, PASDA employs a set of classification heuristics to produce a best effort equivalence classification instead. To further improve the utility of its results, PASDA aims to provide additional supporting information consisting of execution traces and concrete and symbolic inputs and outputs along with its equivalence classification results.

To evaluate PASDA's equivalence checking accuracy and runtime performance, we used an existing equivalence checking benchmark (Badihi et al., 2020) and compared the results to three state-of-the-art equivalence checking approaches (i.e., DSE (Person et al., 2008), ARDiff (Badihi et al., 2020), and PRV (Böhme et al., 2013)). PASDA was able to provide non-/equivalence proofs for 61%–74% of cases in the benchmark at timeout settings from 10 s to 3600 s, thus achieving equivalence checking accuracies that are 3%–7% higher than the best results achieved by the three existing tools. Furthermore, PASDA's best effort classifications were correct for 70%–75% of equivalent and 55%–85% of non-equivalent cases across the different timeouts.

Our evaluation results demonstrate that PASDA can provide more complete descriptions of analyzed program pairs than existing approaches through (i) its increased equivalence checking accuracy and (ii) the best effort equivalence classifications that are newly introduced by it. Increases in equivalence checking accuracy, i.e., in the percentage of cases that can be proven to be non-/equivalent, directly benefit existing use cases that rely on equivalence checking such as test amplification (Danglot et al., 2019) and refactoring assurance for developers (Person et al., 2008). Since best effort equivalence classifications always arise as the result of partial non-/equivalence proofs, we envision that they will benefit use cases such as test case prioritization, fault localization, and general support for manual debugging tasks. All these tasks benefit from more complete descriptions of program behaviors even in the absence of full non-/equivalence proofs.

In our future work, we plan to conduct more detailed investigations of the benefits that can be derived from the information that PASDA provides in the context of the aforementioned use cases. Especially in the manual debugging context, further studies with developers are needed to identify (i) which parts of PASDA's information are useful for developers during different steps of the debugging process and (ii) how to appropriately present this information to them.

CRediT authorship contribution statement

Johann Glock: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Josef Pichler:** Conceptualization, Supervision, Writing – review & editing. **Martin Pinzger:** Conceptualization, Funding acquisition, Supervision, Writing – review & editing, Resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A replication package that contains our implementation, the used benchmark cases, our evaluation scripts, and all evaluation results is available on Zenodo (https://doi.org/10.5281/zenodo.7595851).

Acknowledgments

This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/P36698. For open access purposes, the author has applied a CC BY public copyright license to any author accepted manuscript version arising from this submission.

References

Backes, J., Person, S., Rungta, N., Tkachuk, O., 2013. Regression verification using impact summaries. In: Proceedings of the 20th International SPIN Workshop on Model Checking Software. Springer, pp. 99–116. http://dx.doi.org/10.1007/978-3-642-39176-7_7.

Badihi, S., Akinotcho, F., Li, Y., Rubin, J., 2020. ARDiff: Scaling program equivalence checking via iterative abstraction and refinement of common code. In: Proceedings of the 28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 13–24. http://dx.doi.org/10.1145/3368089.3409757.

Badihi, S., Akinotcho, F., Li, Y., Rubin, J., 2022. GitHub - resess/ARDiff. URL https://github.com/resess/ARDiff.

Baldoni, R., Coppa, E., D'elia, D., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 1–39. http://dx.doi.org/10.1145/3182657.

Barthe, G., Crespo, J.M., Kunz, C., 2011. Relational verification using product programs. In: Proceedings of the 17th International Symposium on Formal Methods. Springer, pp. 200–214. http://dx.doi.org/10.1007/978-3-642-21437-0_17.

Beckert, B., Ulbrich, M., 2018. Trends in relational program verification. In: Principled Software Development. Springer, pp. 41–58. http://dx.doi.org/10.1007/978-3-319-98047-8_3.

Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P., 2013. Precision reuse for efficient regression verification. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. ACM, pp. 389–399. http://dx.doi.org/10.1145/2491411.2491429.

Böhme, M., Oliveira, B.C., Roychoudhury, A., 2013. Partition-based regression verification. In: Proceedings of the 35th International Conference on Software Engineering. IEEE, pp. 302–311. http://dx.doi.org/10.1109/ICSE.2013.6606576.

Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C., Sen, K., Tillmann, N., Visser, W., 2011. Symbolic execution for software testing in practice: Preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering. IEEE, pp. 1066–1071. http://dx.doi.org/10.1145/1985793.1985995.

Cadar, C., Sen, K., 2013. Symbolic execution for software testing: Three decades later. Commun. ACM 56 (2), 82–90. http://dx.doi.org/10.1145/2408776.2408795.

Cousot, P., Cousot, R., 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th Symposium on Principles of Programming Languages. ACM, pp. 238–252. http://dx.doi.org/10.1145/512950.512973.

Dahiya, M., Bansal, S., 2017. Black-box equivalence checking across compiler optimizations. In: Proceedings of the 15th Asian Symposium on Programming Languages and Systems. Springer, pp. 127–147. http://dx.doi.org/10.1007/978-3-319-71237-6_7.

Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., Baudry, B., 2019. A snowballing literature study on test amplification. J. Syst. Softw. 157, http://dx.doi.org/10.1016/j.jss.2019.110398.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: Proceeding of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24.

Di Grazia, L., Bredl, P., Pradel, M., 2022. DiffSearch: A scalable and precise search engine for code changes. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2022.3218859.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M., 2014. Fine-grained and accurate source code differencing. In: Proceedings of the 29th International Conference on Automated Software Engineering. ACM, pp. 313–324. http://dx.doi.org/10.1145/2642937.2642982.

Fedyukovich, G., Gurfinkel, A., Sharygina, N., 2016. Property directed equivalence via abstract simulation. In: Proceedings of the 28th International Conference on Computer Aided Verification. Springer, pp. 433–453. http://dx.doi.org/10.1007/978-3-319-41540-6_24.

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M., 2014. Automating regression verification. In: Proceedings of the 29th International Conference on Automated Software Engineering. ACM, pp. 349–360. http://dx.doi.org/10.1145/2642937.2642987.

Glock, J., Pichler, J., Pinzger, M., 2023. Replication package for: "PASDA: A partition-based semantic differencing approach with best effort classification of undecided cases". http://dx.doi.org/10.5281/zenodo.7595851.

Godefroid, P., Luchaup, D., 2011. Automatic partial loop summarization in dynamic test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, pp. 23–33. http://dx.doi.org/10.1145/2001420.2001424.

Godlin, B., Strichman, O., 2009. Regression verification. In: Proceedings of the 46th Annual Design Automation Conference. ACM, pp. 466–471. http://dx.doi.org/10.1145/1629911.1630034.

Havelund, K., Pressburger, T., 2000. Model checking Java programs using Java PathFinder. Int. J. Softw. Tools Technol. Transfer 2, 366–381. http://dx.doi.org/10.1007/s100090050043.

Idrees, S.Q., Beszédes, Á., 2022. A survey of challenges in spectrum-based software fault localization. IEEE Access 10, 10618–10639. http://dx.doi.org/10.1109/ACCESS.2022.3144079.

Jakobs, M.-C., 2021. PEQcheck: Localized and context-aware checking of functional equivalence. In: Proceedings of the 9th International Conference on Formal Methods in Software Engineering. IEEE, pp. 130–140. http://dx.doi.org/10.1109/FormaliSE52586.2021.00019.

Jakobs, M.-C., Wiesner, M., 2022. PEQtest: Testing functional equivalence. In: Proceedings of the 25th International Conference on Fundamental Approaches to Software Engineering. Springer, pp. 184–204. http://dx.doi.org/10.1007/978-3-030-99429-7_11.

Khatibsyarbini, M., Isa, M.A., Jawawi, D.N.A., Tumeng, R., 2018. Test case prioritization approaches in regression testing: A systematic literature review. Inf. Softw. Technol. 93, 74–93. http://dx.doi.org/10.1016/J.INFSOF.2017.08.014.

Kim, M., Notkin, D., 2009. Discovering and representing systematic code changes. In: Proceedings of the 31st International Conference on Software Engineering. IEEE, pp. 309–319. http://dx.doi.org/10.1109/ICSE.2009.5070531.

King, J., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394. http://dx.doi.org/10.1145/360248.360252.

Kuchta, T., Palikareva, H., Cadar, C., 2018. Shadow symbolic execution for testing software patches. ACM Trans. Softw. Eng. Methodol. 27 (3), 1–32. http://dx.doi.org/10.1145/3208952.

Lahiri, S., Hawblitzel, C., Kawaguchi, M., Rebêlo, H., 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In: Proceedings of the 24th International Conference on Computer Aided Verification. Springer, pp. 712–717. http://dx.doi.org/10.1007/978-3-642-31424-7_54.

LaToza, T.D., Myers, B.A., 2010. Hard-to-answer questions about code. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools. ACM, pp. 8:1–8:6. http://dx.doi.org/10.1145/1937117.1937125.

Le, W., Pattison, S., 2014. Patch verification via multiversion interprocedural control flow graphs. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 1047–1058. http://dx.doi.org/10.1145/2568225.2568304.

Mercaldo, F., Santone, A., 2021. Formal equivalence checking for mobile malware detection and family classification. IEEE Trans. Softw. Eng. 48 (7), 2643–2657. http://dx.doi.org/10.1109/TSE.2021.3067061.

Mercer, E., Person, S., Rungta, N., 2012. Computing and visualizing the impact of change with Java PathFinder extensions. ACM SIGSOFT Softw. Eng. Not. 37 (6), 1–5. http://dx.doi.org/10.1145/2382756.2382801.

Mora, F., Li, Y., Rubin, J., Chechik, M., 2018. Client-specific equivalence checking. In: Proceedings of the 33rd International Conference on Automated Software Engineering. ACM, pp. 441–451. http://dx.doi.org/10.1145/3238147.3238178.

Noller, Y., Păsăreanu, C., Böhme, M., Sun, Y., Nguyen, H.L., Grunske, L., 2020. HyDiff: Hybrid differential software analysis. In: Proceedings of the 42nd International Conference on Software Engineering. ACM, pp. 1273–1285. http://dx.doi.org/10.1145/3377811.3380363.

Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers? In: Proceedings of the 20th International Symposium on Software Testing and Analysis. ACM, pp. 199–209. http://dx.doi.org/10.1145/2001420.2001445.

Partush, N., Yahav, E., 2014. Abstract semantic differencing via speculative correlation. In: Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages & Applications. ACM, pp. 811–828. http://dx.doi.org/10.1145/2660193.2660245.

Păsăreanu, C., Rungta, N., 2010. Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of the 25th International Conference on Automated Software Engineering. ACM, pp. 179–180. http://dx.doi.org/10.1145/1858996.1859035.

Păsăreanu, C., Visser, W., 2009. A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Technol. Transfer 11 (4), http://dx.doi.org/10.1007/s10009-009-0118-1.

Person, S., Dwyer, M., Elbaum, S., Păsăreanu, C., 2008. Differential symbolic execution. In: Proceedings of the 16th International Symposium on Foundations of Software Engineering. ACM, pp. 226–237. http://dx.doi.org/10.1145/1453101.1453131.

Person, S., Yang, G., Rungta, N., Khurshid, S., 2011. Directed incremental symbolic execution. ACM SIGPLAN Not. 46 (6), 504–515. http://dx.doi.org/10.1145/1993316.1993558.

Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V., 2004. Dex: A semantic-graph differencing tool for studying changes in large code bases. In: Proceedings of the 20th International Conference on Software Maintenance. IEEE, pp. 188–197. http://dx.doi.org/10.1109/ICSM.2004.1357803.

Ramos, D., Engler, D., 2011. Practical, low-effort equivalence verification of real code. In: Proceedings of the 23rd International Conference on Computer Aided Verification. Springer, pp. 669–685. http://dx.doi.org/10.1007/978-3-642-22110-1_55.

Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W., 2020. Java Ranger: Statically summarizing regions for efficient symbolic execution of Java. In: Proceedings of the 28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 123—-134. http://dx.doi.org/10.1145/3368089.3409734.

Sun, H., Zhang, X., Zheng, Y., Zeng, Q., 2016. IntEQ: Recognizing benign integer overflows via equivalence checking across multiple precisions. In: Proceedings of the 38th International Conference on Software Engineering. IEEE, pp. 1051–1062. http://dx.doi.org/10.1145/2884781.2884820.

The Apache Software Foundation, 2007. StopWatch (Apache Commons Lang 2.4 API). URL https://commons.apache.org/proper/commons-lang/javadocs/api-2.4/org/apache/commons/lang/time/StopWatch.html.

The Java PathFinder Contributors, 2005. GitHub: Java PathFinder (JPF). URL https://github.com/javapathfinder/jpf-core.

Winter, E., Bowes, D., Counsell, S., Hall, T., Haraldsson, S.O., Nowack, V., Woodward, J.R., 2022. How do developers really feel about bug fixing? Directions for automatic program repair. IEEE Trans. Softw. Eng. 49 (4), 1823–1841. http://dx.doi.org/10.1109/TSE.2022.3194188.

Wong, E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740. http://dx.doi.org/10.1109/TSE.2016.2521368.

Wu, X., Li, M., Li, Y., 2021a. EvoMe: A software evolution management engine based on differential factbase. In: Proceedings of the 36th International Conference on Automated Software Engineering. ACM, pp. 1252–1256. http://dx.doi.org/10.1109/ASE51524.2021.9678795.

Wu, X., Zhu, C., Li, Y., 2021b. Diffbase: A differential factbase for effective software evolution management. In: Proceedings of the 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 503–515. http://dx.doi.org/10.1145/3468264.3468605.

Yang, G., Person, S., Rungta, N., Khurshid, S., 2014. Directed incremental symbolic execution. ACM Trans. Softw. Eng. Methodol. 24 (1), http://dx.doi.org/10.1145/1993316.1993558.

Johann Glock is a Ph.D. student in the Software Engineering Research Group (SERG) at the University of Klagenfurt, Austria. He received his M.Sc. in software engineering at the University of Applied Sciences Upper Austria, Campus Hagenberg, Austria in 2020. His research interests are in program analysis, program comprehension, and formal methods.

Josef Pichler is a professor of programming and project development at the University of Applied Sciences Upper Austria, Campus Hagenberg, Austria. His research interests include static code analysis, reverse engineering, software quality, and software maintenance. He has 20 years of experience in software engineering research. Pichler teaches courses on algorithms, programming, software verification, software maintenance and evolution, and requirements engineering. He earned his Ph.D. in computer science from Johannes Kepler University Linz, Austria.

Martin Pinzger is a full professor at the University of Klagenfurt, Austria where he is heading the Software Engineering Research Group (SERG). His research interests are in software evolution, mining software repositories, program analysis, software visualization, and automating software engineering tasks. He is a member of ACM and a senior member of IEEE.