Practical Inference of Nullability Types

Nima Karimipour*
nima.karimipour@email.ucr.edu
University of California, Riverside
Riverside, California, USA

Justin Pham
jpham079@ucr.edu
University of California, Riverside
Riverside, California, USA

Lazaro Clapp
lazaro@uber.com
Uber Technologies
San Francisco, California, USA

Manu Sridharan
manu@cs.ucr.edu
University of California, Riverside
Riverside, California, USA

ABSTRACT

`NullPointerException`s (NPEs), caused by dereferencing `null`, frequently cause crashes in Java programs. Pluggable type checking is highly effective in preventing Java NPEs. However, this approach is difficult to adopt for large, existing code bases, as it requires manually inserting a significant number of type qualifiers into the code. Hence, a tool to automatically infer these qualifiers could make adoption of type-based NPE prevention significantly easier.

We present a novel and practical approach to automatic inference of nullability type qualifiers for Java. Our technique searches for a set of qualifiers that maximizes the amount of code that can be successfully type checked. The search uses the type checker as a black box oracle, easing compatibility with existing tools. However, this approach can be costly, as evaluating the impact of a qualifier requires re-running the checker. We present a technique for safely evaluating many qualifiers in a single checker run, dramatically reducing running times. We also describe extensions to make the approach practical in a real-world deployment.

We implemented our approach in an open-source tool NullAwayAnnotator, designed to work with the NullAway type checker. We evaluated NullAwayAnnotator's effectiveness on both open-source projects and commercial code. NullAwayAnnotator reduces the number of reported NullAway errors by 69.5% on average. Further, our optimizations enable NullAwayAnnotator to scale to large Java programs. NullAwayAnnotator has been highly effective in practice: in a production deployment, it has already been used to add NullAway checking to 160 production modules totaling over 1.3 million lines of Java code.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

pluggable type systems, null safety, static analysis, inference

*This work was partially completed during an internship at Uber Technologies Inc.

ACM Reference Format:
Nima Karimipour, Justin Pham, Lazaro Clapp, and Manu Sridharan. 2023. Practical Inference of Nullability Types. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616326

1 INTRODUCTION

`NullPointerException`s (NPEs), caused by a dereference of `null`, are a well-known and common cause of crashes in Java programs. Hence, there has been a great deal of past research on preventing null dereferences (e.g., [17, 22, 24, 28]). Type-based approaches to nullness checking are growing in popularity. In this approach, types include information on whether each expression may evaluate to `null`, and only expressions that cannot be `null` can be dereferenced. Recent languages like Kotlin [21] and Swift [35] build null safety into their type systems. Further, pluggable type checkers like the Checker Framework [9, 28], Eradicate [10], Nullsafe [31], and NullAway [4] leverage type qualifiers [13] to add type-based nullness checking to Java.

Type-based nullness checking can be difficult to adopt for existing Java code bases, due to the need to manually add type qualifiers into the code. To enable incremental and modular checking, type-based nullness checkers require explicit nullability annotations on field, parameter, and return types. Hence, manual effort is required to annotate any existing program for a type-based checker. While default assumptions for unannotated types reduces the annotation burden considerably [28], a significant number of explicit annotations must still be written.[1] Our goal is to develop a tool for automatically *inferring* nullability qualifiers for existing code, thereby dramatically easing adoption of type-based nullness checking.

We desire a practical inference tool that can be used with real-world code bases and type checkers, and as such have three key requirements. First, we require a tool that provides a "best effort" partial solution in cases where a program cannot be verified without code changes. Existing code may have real bugs, or may be correct but written in a style that is not amenable to type-based verification. In such cases, an inference tool can still provide significant value by adding annotations that enable type checking of *most* of the original program. This way, most newly-added or modified code (very often

[1]Banerjee et al. [4] report an average of roughly 13 explicit annotations per KLoC on their open-source benchmarks, ranging up to 46 annotations per KLoC.

the source of defects [26]) will be type checked, while developers can gradually adopt and enable checking for the remaining code.

Second, we require the inference tool to work alongside an existing type checker implementation. Inference approaches often require a type checker that supports both checking and inference directly, e.g., using constraint generation and solving. Our tool infers annotations for the existing production-quality NullAway type checker [4], which does not use constraints. While the constraint-based approach is elegant and efficient, re-implementation of a complex checker like NullAway to use constraints would require a huge effort. For example, NullAway employs *ad hoc* handling of various coding patterns and libraries (e.g., stream libraries and gRPC [15]) to reduce false positives [27], and this customized handling logic would have to be encoded precisely using constraints. To avoid re-implementation, we desire an inference approach that primarily treats the type-checking tool as a black-box oracle, relying only on its reported errors to perform inference.

Third, we require the inference tool to be performant. For our use cases, the tool must be able to run in an overnight job (roughly 8 hours maximum). With a longer running time, deployment of the tool becomes less practical, due to the compute resources required, and the fact that the target code could be changing frequently. We are unaware of any existing approach that meets these three requirements.

In this paper, we present a novel approach to nullability type inference suited to these requirements. Given an unannotated program, our approach searches for a set of type qualifiers that minimizes the remaining number of NullAway errors, thereby maximizing the amount of code NullAway is able to type check. Finding a good set of qualifiers is non-trivial; we found that eagerly inserting all possible qualifiers could *increase* the final number of errors. Our strategy evaluates candidate qualifiers using a bounded-depth search, iterated to a fixed point.

A naïve implementation of our search strategy is too slow for large code bases. When treating the type checker as a black-box oracle, evaluating the impact of a candidate qualifier on the error count requires re-running the type checker. Though NullAway is highly optimized [4], each run still requires a re-compilation of the code and can take tens of seconds or longer. For larger programs, a straightforward version of our search required running over 1,000 builds, making the tool too slow.

Our key insight is that many qualifiers impact the error count *independently* of each other. Since NullAway performs modular checking, the code regions where new errors may appear due to qualifier insertion are localized and can be computed ahead of time. Given this information, we construct a graph representing which candidate qualifiers may "conflict" by causing new errors in an overlapping region. Then, we use graph coloring to find sets of non-conflicting qualifiers, which can all be tested simultaneously within a single NullAway build. With this approach, many fewer runs of NullAway are required, dramatically reducing running time and making the tool practical.

We also describe extensions to our technique for handling real-world code patterns and deployment. We show how our algorithm can incorporate usage information from client code when annotating a library, significantly easing deployment in a large, modular code base. And, we describe a specialized handling of field initialization to better handle certain common patterns.

```
1  class Test {
2    +@Nullable Object f1 = null;
3    +@SuppressWarnings("NullAway") Object f2 = null;
4    +@Nullable Object f3 = null;
5    +@Nullable Object f4 = null;
6    +@Nullable Object f5 = f4;
7    String m1() {
8      return f1 != null ? f1.toString() : f2.toString();
9    }
10   int m2() {
11     return f3 != null ? f3.hashCode() : f2.hashCode();
12   }
13   +@Nullable Object m3() {
14     return f4;
15   }
16 }
```

Figure 1: Motivating example for inference. Green text indicates where annotations are inserted by our technique. Our tool chooses to suppress the error on line 3, to maximize the amount of code checked by NullAway.

We implemented our approach in an open-source tool NullAwayAnnotator, which generates annotations suitable for directly enabling NullAway checking. We performed an extensive empirical evaluation, on both open-source projects and a set of commercial code modules at Uber Technologies Inc. (Uber). The evaluation showed that NullAwayAnnotator decreased the final number of errors reported by NullAway by an average of 69.5% (36.9%–90.1%). Further, our optimizations were critical for acceptable performance, reducing running time by an average of 6.1X (2.0X–17.8X) and eliminating two timeouts. NullAwayAnnotator is deployed at Uber for direct use by developers, and it has been highly effective: it has been used to enable NullAway checking for 160 production modules, totaling over 1.3 million lines of code.

This paper makes the following key contributions:

- We present a technique to infer type qualifiers that make as much of an existing program as possible verifiable by NullAway, while treating NullAway as a black box oracle.
- We describe an optimized search that leverages conflict detection via graph coloring to simultaneously evaluate many candidate qualifiers, reducing running time.
- We present an open-source implementation of the approach, NullAwayAnnotator, and show its effectiveness in an extensive experimental evaluation.

2 OVERVIEW

In this section, we give an overview of our nullability inference technique and present relevant background. We illustrate our technique using the example code in Figure 1.

2.1 Type-Based Nullness Checking

We briefly introduce key ideas of type-based nullness checking; see the literature for details [4, 9, 28]. Type-based nullness checkers for Java use type qualifiers [13] to capture whether a type includes or excludes `null`. These qualifiers are written in source code using Java annotations, prefixed with `@`. Such checkers typically use `@Nullable` to qualify a type that contains `null`, and `@NonNull` for a type that does not. An unqualified type is treated as `@NonNull` by default, except for local variables, whose qualifiers are inferred automatically [28].

Given these qualifiers, type checking guarantees that a `@Nullable` expression is never assigned to a `@NonNull` location, and that a `@Nullable` expression is never dereferenced. Assuming object fields are properly initialized, and that all executing code has been type checked, these properties together guarantee the program will be free of NPEs. For clarity, we defer discussing the impacts of field initialization checking to Section 6, and for now assume that all `@NonNull` fields are appropriately initialized in a constructor.

Consider the code example in Figure 1, ignoring the green inserted annotations. Running NullAway on this code will yield four errors, one for each of lines 2 to 5, since each line assigns `null` to a `@NonNull` field (`@NonNull` by default since they are unannotated). These errors can be removed by changing the type qualifier of each field to be `@Nullable`, e.g., writing `@Nullable Object f2` on line 3. However, adding these qualifiers can lead to new NullAway errors; e.g., making `f2` `@Nullable` causes a new error on line 8, since `f2.toString()` then dereferences a `@Nullable` expression. Currently, adding type-based nullness checking to an existing code base requires repeatedly adding annotations and changing code manually until no errors remain, a tedious, time-consuming process.

To ensure null safety, type-based nullness checkers must also enforce standard subtyping rules for method overriding, i.e., covariant return types and contravariant parameter types. So, if a method `Super.m1(p)` has a `@NonNull` return type and a `@Nullable` parameter type, an overriding method `Sub.m1(p)` (where `Sub extends Super`) must *not* have a `@Nullable` return type or a `@NonNull` parameter type; see further discussion in the literature [4, 28].

2.2 Inference Approach

The goal of our work is to automate a significant portion of the work required to adopt type-based nullness checking for pre-existing, real-world code bases. We focus on the problem of inferring a set of `@Nullable` type qualifiers that minimizes the number of remaining errors reported by NullAway. As noted in Section 1, we require inference to use NullAway as a black-box oracle. Hence, checking the impact of a set of qualifiers on NullAway's error count requires re-running NullAway on a modified version of the program with the qualifiers inserted. We only attempt to address errors fixable via qualifier insertion, i.e., errors stemming from either assigning a `@Nullable` expression into a `@NonNull` location or an incorrect method override (*not* dereferences of `@Nullable` expressions). From such errors, we create a set of candidate *fixes* based on qualifier insertion, and then use NullAway to test if those fixes reduce the overall error count.

Determining the impact of a candidate fix may require multiple iterations, as fixes can cause new errors which themselves are amenable to fixing. For Figure 1, adding a `@Nullable` qualifier to fix the error at line 3 leads to two new errors at line 8 and line 11, both due to dereference of a `@Nullable` expression. Since our approach cannot fix these new errors, the line 3 fix increases the overall error count and is not retained. It makes sense to leave this error for the developer to handle, as the lack of null checks for `f2` on lines 8 and 11 contradict the initialization of `f2` to `null`, and it is unclear how to automatically resolve this contradiction.

Adding a `@Nullable` qualifier for the error on line 5 also causes two new errors, on line 6 and line 14. However, these new errors can be addressed via two more `@Nullable` qualifiers (on line 6 and the return type on line 13), and these qualifiers cause no further errors, yielding an overall decrease. Our search discovers the final solution shown in Figure 1, which includes these three qualifiers. We detail our iterative search strategy in Section 3.

To create a code change that can be adopted immediately, our tool suppresses any remaining NullAway errors after its search is complete; for Figure 1, we insert the `@SuppressWarnings` annotation on line 3. Adding suppressions is not ideal, since they may mask real NPE bugs remaining in the code. However, a key benefit of this approach is that after remaining errors are suppressed, NullAway can be enabled *for all future builds* of the code. Developers then benefit from NullAway checking for any subsequent code change outside of a suppressed region or any newly-written code, and recent code changes are often the source of defects [26]. Remaining suppressions can be removed gradually as part of periodic code cleanup efforts. Also note that with this approach, the initial code change enabling NullAway introduces *no semantic changes* (easing code review) and can be generated *with no manual effort*. Minimizing the number of remaining NullAway errors during inference maximizes the amount of code subject to NullAway checking after the inferred qualifiers are adopted.

2.3 Optimizing Performance

A naïve implementation of our search runs too slowly in practice, due to the cost of repeatedly running NullAway to evaluate candidate solutions. Reducing running time therefore requires reducing the number of NullAway runs required for inference. This reduction could be achieved if multiple *independent* qualifiers could be evaluated *simultaneously* in a single run of NullAway. Two qualifiers are independent if any NullAway errors removed or caused by each qualifier are guaranteed to be in non-overlapping regions of code. Achieving a speedup requires efficiently computing large groups of independent qualifiers.

Our first key insight was that due to modular type checking, the potential code regions where a `@Nullable` qualifier may add or remove errors can be computed precisely and cheaply. For example, a `@Nullable` qualifier on a field can only impact error counts within methods that read or write the field; new errors *cannot* appear other unrelated methods. So, for the initial errors in Figure 1, our approach determines that the potentially impacted regions for `@Nullable` qualifiers are `m1` for `f1`, `m1` and `m2` for `f2`, `m2` for `f3`, and `m3` for `f4`.[2]

Intuitively, many qualifiers in a large program may be independent (e.g., when they apply to private state of distinct classes), but a question remains as to how to quickly find large sets of independent

[2]Each field initializer (e.g. the right hand side of `Object f5 = f4`) is also a region, which we ignore here for simplicity; see Section 4.1 for details.

Algorithm 1 Pseudocode for unoptimized search.

```
1:  procedure FindNullableQualifiers($P, d$)
2:      $E \leftarrow$ NullAwayErrors($P$)
3:      $F_{all} \leftarrow$ FixLocations($E$)
4:      $F_{new} \leftarrow F_{all}$
5:      while $F_{new} \neq \emptyset$ do
6:          $F_{good} \leftarrow$ EvaluateFixes($P, F_{new}, F_{all}, d, E$)
7:          $P \leftarrow$ ApplyFixes($P, F_{good}$)
8:          $E \leftarrow$ NullAwayErrors($P$)
9:          $F \leftarrow$ FixLocations($E$)
10:         $F_{new} \leftarrow F - F_{all}$
11:         $F_{all} \leftarrow F_{all} \cup F_{new}$
12:     end while
13:     return $P$
14: end procedure
15: procedure EvaluateFixes($P, F, F_{all}, d, E$)
16:     if $d = 0$ return $F$
17:     $F_{good} \leftarrow \emptyset$
18:     for $f \in F$ do
19:         $curFixes \leftarrow \{f\}$
20:         for $i \in [1, d]$ do
21:             $P' \leftarrow$ ApplyFixes($P, curFixes$)
22:             $E' \leftarrow$ NullAwayErrors($P'$)
23:             $F' \leftarrow$ FixLocations($E'$)
24:             if $|E| - |E'| \geq 0$ then
25:                 $F_{good} \leftarrow F_{good} \cup \{curFixes\}$
26:                 break
27:             end if
28:             $newFixes \leftarrow F' - F_{all}$
29:             if $newFixes = \emptyset$ then
30:                 break
31:             else
32:                 $curFixes \leftarrow curFixes \cup newFixes$
33:             end if
34:         end for
35:     end for
36:     return $F_{good}$
37: end procedure
```

qualifiers. Our second key insight was that groups of independent fixes could be computed efficiently via graph coloring [18]. In compilers, graph coloring is often used for register allocation [2]. In our scenario, two fixes are independent if their impacted regions do not overlap. We construct a graph representation where nodes represents sets of fixes, and an edge between two nodes reflects overlapping regions for their fixes. This graph can be colored to find groups of fixes that can be evaluated in a single NullAway run. For our example, the `f1`, `f3`, and `f4` fixes for Figure 1 can all be evaluated in a single run of NullAway (see Figure 2 in Section 4). Overall, the optimized approach reduces the number of NullAway runs required by two for Figure 1, and as shown in Section 8, the reductions for real-world code are much more dramatic.

3 SEARCH-BASED QUALIFIER INFERENCE

In this section, we present an unoptimized version of our inference algorithm for `@Nullable` qualifiers, to make clear how our technique explores and evaluates the space of possible qualifiers. In Section 4, we present our graph-coloring-based optimized search.

Pseudocode for the unoptimized technique appears in Algorithm 1. Given an unannotated program P and a depth limit d for evaluating fixes, FindNullableQualifiers(P, d) returns a modified program with `@Nullable` qualifiers that reduce the number of reported NullAway errors. It uses a procedure EvaluateFixes that, given a program P, a set of candidate fixes F, all previously-considered fixes F_{all}, the depth limit d, and the NullAway errors E for P, returns a set F_{good} of fixes that reduce the NullAway error count for P.

Both procedures make use of three key subroutines (whose implementations are not shown). NullAwayErrors runs NullAway to compute the errors it reports for a program P. Given a set of errors E, FixLocations(E) first determines the subset of errors in E that can be fixed via `@Nullable` insertion (see Section 2.2). For that subset, it returns a set containing *fixes* for each error, i.e., the code locations where `@Nullable` should be inserted to fix the error (multiple locations may be required for a single initializer error; see Section 6). Finally, given a program and a set of fixes, ApplyFixes returns a new program with the fixes inserted.

The algorithm proceeds as follows. FindNullableQualifiers runs a fixed-point loop (lines 5–12), inserting fixes determined by EvaluateFixes to reduce the error count until no new fixes can be found. In EvaluateFixes, if the depth limit $d = 0$, all fixes are assumed to be good (line 16), leading to their eager insertion. Otherwise, for each fix $f \in F$, the algorithm iteratively applies f and any new fixes discovered after applying f up to some maximum depth d (lines 18–35). In the first iteration, only f itself is tested (line 19), with newly discovered fixes added to the *curFixes* set for each subsequent iteration (lines 28–33).

A fix f is determined to be good if the *curFixes* set of fixes for f *does not increase* the NullAway error count (lines 24–27). We keep fixes even if they yield the same error count to improve handling of *fix chains* at lower depth limits. A fix chain occurs when inserting fix f_1 causes a single new error with fix f_2, f_2 leads to f_3, and so on, where the final fix causes no new error. If good fixes needed to strictly reduce the error count, then discovering the goodness of f_1 would require setting d to at least the length of the chain, reducing performance. With our approach, the fix chain is applied even with depth limit 1, due to the outer fixed-point loop.

Algorithm 1 is guaranteed to terminate. In an execution of FindNullableQualifiers, the F_{all} set of fixes grows monotonically, and number of possible fixes for a program is finite. So, eventually, F_{new} must become empty after line 10, causing the loop to terminate.

Example. Consider applying Algorithm 1 to the Figure 1 example, with a depth limit $d = 2$. In the first iteration, $F_{new} = \{2, 3, 4, 5\}$, representing the fix locations for the initial errors reported by NullAway. In EvaluateFixes, the fixes 2 and 4 will be labeled as good at depth 1, since they immediately reduce the error count by 1. Fix 3 is not labeled as good, as it introduces errors that cannot be fixed with `@Nullable` annotations, but fix 5 is labeled good after exploring to depth 2 and applying subsequent fixes at lines 6 and 13 (see Section 2.2 discussion). So, $F_{good} = \{2, 4, 5, 6, 13\}$ in the first fixed-point loop iteration. No new fixes are observed after applying F_{good}, so the algorithm converges, yielding the final set of `@Nullable` qualifiers shown in Figure 1. For this example, the unoptimized algorithm requires 7 calls to NullAwayErrors to compute the final solution; Section 4 will show how we can reduce this number.

4 OPTIMIZED SEARCH

The search algorithm of Section 3 runs too slowly for larger code bases, due to running a large number of NullAway builds. Here, we present optimizations to significantly reduce this cost, based on evaluating multiple independent qualifiers simultaneously in a build. We first define the *potentially-impacted regions* of a qualifier (Section 4.1), used to determine when qualifiers can be evaluated simultaneously. Then, we give details of our graph-coloring-based search algorithm (Section 4.2).

4.1 Potentially-Impacted Regions

Our optimizations exploit the fact that for a type-based nullness checker, introducing a `@Nullable` qualifier on an entity (field, method parameter, or method return) can only impact the final error count in regions of the code *where the entity is directly used or overridden.* This fact stems from the modular nature of the type checking; since no inter-procedural analysis is performed by the checker to determine nullability, the impact of changed nullability for an entity cannot propagate beyond the procedures using the entity.

A *region* is a method, an initializer expression (e.g., the read of `f4` at line 6 in Figure 1), or an initializer block. Given an entity e, the *potentially-impacted regions* for e are the set of regions whose contained error count may change as the result of adding a `@Nullable` qualifier to e. We define *potentially-impacted regions* for e as follows:

- If e is a field f, any region containing a read or write of f is potentially-impacted.
- If e is a parameter or return of method $C.m$, then $C.m$ itself is potentially impacted, as is any region that calls $C.m$. Further, any method that overrides or is overridden by $C.m$ is also potentially impacted.

Regions that read e may contain new errors, since once e is `@Nullable` it cannot be dereferenced or assigned to a `@NonNull` location. Similarly, errors may be removed from regions that write e, since writes of `@Nullable` values into e become legal. Overriding or overridden methods may be impacted due to NullAway's subtyping checks (see Section 2.1). Potentially-impacted regions can be computed using standard type-checking information, like a type hierarchy and symbol tables.

4.2 Optimized Algorithm

Algorithm 2 gives pseudocode for the key EvaluateFixesOpt procedure of our optimized search. The overall algorithm retains the FindNullableQualifiers procedure from Algorithm 1, with the call to EvaluateFixes at line 6 replaced with a call to EvaluateFixesOpt.

EvaluateFixesOpt computes groups of independent fixes using a *conflict graph.* Each node in the graph represents a pair of a root fix f and related fixes *curFixes* being tested, the same state tracked for each fix by the main loop of EvaluateFixes in Algorithm 1. Given a set of nodes, the ConflictGraph procedure constructs a conflict graph, which contains an (undirected) edge between nodes n_1 and n_2 iff the potentially-impacted regions for the fixes in $n_1.curFixes$ and $n_2.curFixes$ overlap. With this representation, the fixes for any two *non-adjacent* nodes are independent, and the impacts of all their fixes may be evaluated simultaneously.

Algorithm 2 Optimized search based on graph coloring.

```
1:  procedure EvaluateFixesOpt($P, F, F_{all}, d, E$)
2:      if $d = 0$ return $F$
3:      $F_{good} \leftarrow \emptyset$
4:      $initNodes \leftarrow \{\langle root : f, curFixes : \{f\}\rangle \mid f \in F\}$
5:      $G \leftarrow$ ConflictGraph($initNodes$)
6:      for $i \in [1, d]$ do
7:          $groups \leftarrow$ Color($G$)
8:          $toRemove \leftarrow \emptyset$
9:          for $S \in groups$ do
10:             $fixes \leftarrow \bigcup_{n \in S} n.curFixes$
11:             $P' \leftarrow$ ApplyFixes($P, fixes$)
12:             $E' \leftarrow$ NullAwayErrors($P'$)
13:             for $n \in S$ do
14:                 $E_n \leftarrow \bigcup_{r \in \text{Regions}(n)}$ ErrorsInRegion($E, r$)
15:                 $E'_n \leftarrow \bigcup_{r \in \text{Regions}(n)}$ ErrorsInRegion($E', r$)
16:                 $F'_n \leftarrow$ FixLocations($E'_n$)
17:                 if $|E_n| - |E'_n| \geq 0$ then
18:                     $F_{good} \leftarrow F_{good} \cup \{n.curFixes\}$
19:                     $toRemove \leftarrow toRemove \cup \{n\}$
20:                 end if
21:                 $newFixes \leftarrow F'_n - F_{all}$
22:                 if $newFixes = \emptyset$ then
23:                     $toRemove \leftarrow toRemove \cup \{n\}$
24:                 else
25:                     $n.curFixes \leftarrow n.curFixes \cup newFixes$
26:                 end if
27:             end for
28:         end for
29:         $G \leftarrow$ ConflictGraph($G.nodes - toRemove$)
30:     end for
31:     return $F_{good}$
32: end procedure
33: procedure ConflictGraph($N$)
34:     $conflictEdges \leftarrow \{(n_1, n_2) \mid n_1, n_2 \in N$
35:                     $\wedge\ \text{Regions}(n_1) \cap \text{Regions}(n_2) \neq \emptyset\}$
36:     return $\langle nodes : N, edges : conflictEdges\rangle$
37: end procedure
38: procedure Regions($n$)
39:     return $\bigcup_{f \in n.curFixes}$ potentially-impacted regions for $f$
40: end procedure
```

After constructing an initial conflict graph (line 5), EvaluateFixesOpt proceeds by iterating up to the depth limit (starting at line 6), similar to the line 20 loop in EvaluateFixes in Algorithm 1. In each iteration, the algorithm computes sets of independent fixes by coloring the current conflict graph (line 7). The Color routine returns a set of sets of nodes *groups*, where for each set $S \in groups$, no pair of nodes in S is adjacent in G. Hence, fixes corresponding to all nodes in S can be evaluated simultaneously in a single NullAway run. Our implementation uses a greedy coloring algorithm [18], which produces sufficiently good results for our needs.

For each group S (line 9), we apply *all* current fixes for all nodes in that group simultaneously, and run NullAway to compute an updated set of errors (lines 10 to 12). This yields a single new set of errors E', from which we must extract the errors specific to each node n, to determine if n's fixes reduce the error count. We can do so by finding original and new errors only in the impacted regions for n (lines 14 and 15), which are guaranteed not to overlap with any other node in S. The logic in lines 16–26 is analogous to that in lines 23–33 in Algorithm 1, except that instead of breaking out of the loop when a fix is fully handled, the corresponding node is marked for removal from the conflict graph. After processing all groups, we recompute the conflict graph (line 29) after removing marked nodes, and continue to the next depth level. Recomputing the conflict graph is needed even if no nodes are removed, as the list of current fixes for each node may have changed, impacting the required edges between nodes. In the end, EvaluateFixesOpt returns the same result as EvaluateFixes from Algorithm 1 for the same inputs, but with a smaller number of calls to NullAwayErrors.

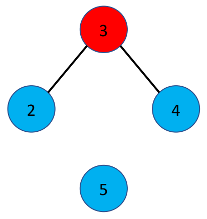

Figure 2: Conflict graph in first iteration of Algorithm 2 run on Figure 1.

Example. Consider applying Algorithm 2 to the Figure 1 example, again with $d = 2$. Figure 2 shows the initial conflict graph for the example, capturing which fixes have overlapping impacted regions. The coloring of Figure 2 shows that fixes 2, 4, and 5 can all be evaluated with a single call to NullAwayErrors. Overall, the same final result is computed, but with 5 calls to NullAwayErrors as compared to 7 for Algorithm 1; real-world improvements are more dramatic (see Section 8).

5 CLIENT CODE

Real-world code bases are often composed of many inter-dependent modules, to enable more scalable development with large teams. For scalability and ease of review, it would be very useful to be able to apply nullability inference one module at a time for such projects. As described thus far, our inference algorithm accounts for NullAway errors within the code being annotated, which we term the *target* code. In a multi-module scenario, the target code would be the single module being annotated. However, our presented technique does not yet account for potential NullAway errors in other modules dependent on the module being annotated, which we refer to as *client* code. Ignoring client code can lead to undesirable inference results. For example, if a public method `m` has no calls from within target code, inference will assume that making `m`'s return type `@Nullable` will not introduce any new errors, even if client code assumes `m` does not return `null`. Here we describe an inference extension to account for assumptions made by clients of the target code, crucial for usability in practice.

A naïve approach to handling client code could simply compile all client code alongside the target in each build and observe the impact of fixes in clients. We assume annotations cannot be added to client code, so errors should only be treated as fixable if the fix location is in the target. Unfortunately, this approach does not scale to a large amount of client code (even with the optimizations of Section 4), as each individual build becomes much more expensive when all client code is included.

Instead, we handle client code via up-front *caching* of the impacts of fixes on clients. For simplicity, here we assume that client code only interacts with the target via calls to public methods; field accesses and method overriding can be handled similarly. Under this assumption, fixes in the target may impact the client in two ways: (1) making a public return type `@Nullable` may cause new client NullAway errors, and (2) making a public parameter type `@Nullable` may remove client errors. Before our core search, we run up-front builds of client code to find and cache the client impacts for each such fix; we then use the cached information during the search and only build the target code. Empirically, the number of up-front builds required was far fewer than the number of builds run during the search (see Section 8), so overall this caching yields a large speedup.

Our caching phase runs the following builds. First, we build the client code with unmodified target code, and cache all NullAway errors in client code caused by passing a `@Nullable` expression to a `@NonNull` target method parameter (case (2) above). Then, we run builds to cache the client errors introduced by making *each* public target method return `@Nullable` (case (1) above), noting when such errors have a fix location in the target. We dramatically reduce the number of such builds needed for this phase by re-using the graph coloring optimization of Section 4.2. With fully-cached information about relevant client errors and their fix locations, we augment Algorithms 1 and 2 to use this information during the search, requiring no further client code builds.

Result equivalence. The caching scheme described above can be generalized to cache the errors caused by *any* potential fix, whether in target or client code. This more extensive caching could yield further search speedups: when applying a set of fixes F, if the impacts of all fixes in F are already cached, the cached information could be used to compute the overall impact of F, avoiding a run of NullAway. Unfortunately, in certain cases the combined impacts of individual fixes in F are *not* equivalent to the impact of applying all fixes in F together. This occurs because it is possible for a new error to appear only when a *pair* of fixes is introduced, but not when either fix is added individually. Consider this excerpt:

```
class C {
  Object f1 = new Object();
  Object f2 = new Object();
  [...]
  void m() {
    f1 = f2;
    f1.toString();
  }
}
```

Say that due to NullAway errors in other code (not shown), the search considers making `f1` and `f2` `@Nullable`. NullAway reports a dereference-of-`@Nullable` error at the `f1.toString()` call only if *both* `f1` and `f2` are `@Nullable`. If just `f1` is `@Nullable`, no error is reported, since `f2` is `@NonNull` and assigned to `f1` before the call. If just `f2` is `@Nullable`, an error is reported at the `f1 = f2` assignment, but not at the call. Due to such cases, cached information may under-estimate the number of NullAway errors reported when multiple fixes are applied simultaneously. In our evaluation, we never observed unexpected NullAway errors in client code due to this issue.

Weighting. A question remains of how to weight client vs. target NullAway errors during the inference search. E.g., for a small target with a large amount of client code, one may want to weight errors in the target more highly than errors in clients. By default, NullAwayAnnotator weights all errors equally, but different weightings can easily be supported.

At Uber, NullAwayAnnotator is being used to annotate individual modules in a "monorepo" [33] containing hundreds of modules. Some modules are already checked with NullAway, and a key requirement for NullAwayAnnotator is that *no* new NullAway errors should be introduced in other modules when annotating a target. This feature eases code review, since the changes introduced by NullAwayAnnotator are thereby limited to semantics-preserving changes to exactly one module, requiring limited review. Before imposing this requirement, we observed cases where NullAwayAnnotator changes caused new errors in dozens of other modules: the fixes for these errors required reviews from many teams and dramatically slowed adoption.

For this scenario, NullAwayAnnotator includes a *strict mode*, in which it only adds annotations to the target if they do not cause new errors in other modules. (In essence, strict mode applies an infinite weight to client errors.) Strict mode may increase the final number of remaining NullAway errors in the target. However, it makes adoption of changes from NullAwayAnnotator much easier, and as noted in Section 2.2, getting these changes merged quickly provides an immediate benefit, as new code and most code modifications in the target then benefit from NullAway checking.

6 INITIALIZATION

Beyond the checks discussed so far, NullAway also checks for correct object initialization. Consider the example of Figure 3. The `t1`, `t2`, and `t3` fields are treated as `@NonNull` by default. However, the `TestInit` constructor fails to initialize the fields, so they could still be `null` at later reads (e.g., if `useFields` were called immediately after the constructor). Hence, NullAway reports an initialization error for the constructor. Our inference technique can handle such a case by inserting a `@Nullable` annotation on all three fields, which removes the error. However, this leads to three new dereference-of-`@Nullable` errors in the `useFields` method, and hence does not decrease the error count.

NullAway also supports *initializer methods* to capture cases where fields are initialized after object construction but before any use [4]. For the Figure 3 example, assume the intended lifecycle of `TestInit` is that after construction, the `init` method should be invoked before any other method in the class. Then, the dereferences in `useFields` are safe, as they will only occur after `init` has run. Such protocols arise regularly in practice, e.g., for Android activities [1]. NullAway treats any method annotated with

```
class TestInit {
  Object t1, t2, t3;
  TestInit() {}
  +@Initializer
  void init(Object o1, Object o2, Object o3) {
    t1 = o1;
    t2 = o2;
    t3 = o3;
  }
  int useFields() {
    // no null checks needed here
    return t1.hashCode() + t2.hashCode()
           + t3.hashCode();
  }
}
```

Figure 3: Example for illustrating initialization checks.

`@Initializer` as a method that runs before all other methods in the class, and reasons about field initialization in such methods appropriately. NullAway does *not* check that client classes actually invoke `@Initializer` methods before other methods, and hence its handling of this feature is unsound [4].

Our approach includes limited support for inferring `@Initializer` annotations. Since NullAway's support for `@Initializer` is unsound, we devised our approach to infer `@Initializer` under narrow conditions, aiming to avoid introducing incorrect annotations. We only add `@Initializer` to a method *m* if the following holds:

(1) *m* must write a `@NonNull` value to at least two otherwise-uninitialized fields, and those fields cannot be overwritten with a `@Nullable` value before *m* returns.
(2) There can be at most one inferred `@Initializer` method per class. If more than one method in *m*'s class meets condition 1, *m* must be the method that initializes the most fields.

Inference of `@Initializer` occurs in a separate pass, before running Algorithm 2, so that the core search does not insert `@Nullable` annotations on fields that are never null once `@Initializer` methods are considered. For the Figure 3 example, our approach would add an `@Initializer` annotation to `init`, thereby removing all errors reported for the class. In our evaluation, we inspected all introduced `@Initializer` annotations to check that they reflected actual lifecycle behavior (see Section 8).

7 IMPLEMENTATION

We have implemented our approach in a tool NullAwayAnnotator, which is open source.[3] To compute potentially-impacted regions for fixes (Section 4.1), we implemented a code structure scanner as an Error Prone plugin checker [12], which runs as part of the Java compilation process (like NullAway). This scanner serializes the relevant information about uses of fields, uses of methods, and the type hierarchy, as computed by the Java compiler. The scanner is implemented in 1,220 (non-blank, non-comment) lines of Java code.

[3] https://github.com/ucr-riple/NullAwayAnnotator

A separate component handles insertion of annotations into source code. It leverages JavaParser [19] to discover annotation insertion locations, and then inserts annotations using string manipulation to ensure whitespace is preserved. Whitespace preservation is critical for making the tool usable in practice, as any unnecessary formatting modifications make changes harder to review. The injector is implemented in 1,522 lines of Java code.

Finally, the core optimized search of NullAwayAnnotator (Section 4.2) is implemented in roughly 4,300 lines of Java code. To run NullAwayAnnotator, a developer must integrate NullAway and the code structure scanner as part of their compilation scripts. NullAwayAnnotator is build-system independent, as a variety of build systems are in common use in the Java ecosystem. For ease of implementation, we made minor modifications to NullAway to serialize its output in a machine-readable format. This output could have been parsed directly from NullAway's error messages, but at greater engineering cost.

8 EVALUATION

In this section, we present an experimental evaluation of NullAwayAnnotator, showing the effectiveness of its inferred annotations and of our performance optimizations.

8.1 Experimental setup and research questions

We evaluated NullAwayAnnotator on two separate datasets of Java code bases. First, we used a collection of 14 open-source Java projects from GitHub. From the most popular Java projects on GitHub (as determined by number of stars), we chose 13 projects that use the Gradle build system [14], to ease integration of NullAway and NullAwayAnnotator's configuration. To ensure greater diversity in the benchmarks, we limited the number of Android projects to five. Finally, we included WALA:Util, a module from the WALA static analysis library [37] maintained by one of the authors, to evaluate adopting NullAway via NullAwayAnnotator on an open-source project (see Section 8.4). We did not create a larger suite of open-source benchmarks due to the manual effort required to integrate NullAway and NullAwayAnnotator into the build scripts of each benchmark.

Second, we used a set of 8 modules from Uber's repository of Java server code. These targets (T1 to T8) were selected on the basis of a one-week sampling of production crash logs. They represented the top-8 targets by NPE count in this dataset, excluding one target on which NullAwayAnnotator crashed and any targets that were already enrolled in NullAway (which may still contain NPEs due to third-party libraries [4]). At the time of our evaluation, targets T1 to T8 were *not* already enrolled onto NullAway using NullAwayAnnotator; we avoided previously-enrolled targets so that all experiments could be run with a single tool version. See Section 8.4 for further discussion regarding real-world usage of NullAwayAnnotator to enroll targets.

Table 1 gives the size and type of each benchmark. While the open-source dataset represents standalone programs and libraries, the Uber dataset consists of build targets: program modules built and unit tested independently, but which serve as part of one or more production services.

Table 1: Benchmark types and sizes, and the error reduction from NullAwayAnnotator at depths 0, 1, and 5.

	Benchmark Name	KLoC	Number of Errors			
			Initial	Depth 0	Depth 1	Depth 5
Open Source Projects	Framework					
	`Conductor`	9.2K	159	170 (+6.9%)	44 (-72.3%)	30 (-81.1%)
	`Mockito`	17.6K	205	95 (-53.7%)	47 (-77.1%)	30 (-85.4%)
	`SpringBoot`	35.1K	777	204 (-73.7%)	184 (-76.3%)	77 (-90.1%)
	Game Engine					
	`LitiEngine`	30.1K	480	468 (-2.5%)	191 (-60.2%)	184 (-61.7%)
	`LibGdx`	92.1K	1549	2314 (+49.4%)	516 (-66.7%)	442 (-71.5%)
	Libraries					
	`MPAndroid`	16.1K	174	489 (+181.0%)	64 (-63.2%)	53 (-69.5%)
	`Glide`	24.6K	287	195 (-32.1%)	112 (-61.0%)	105 (-63.4%)
	`EventBus`	1.9K	49	18 (-63.3%)	12 (-75.5%)	10 (-79.6%)
	`Gson`	8.0K	161	38 (-76.4%)	39 (-75.8%)	28 (-82.6%)
	`Eureka`	8.0K	74	70 (-5.4%)	31 (-58.1%)	25 (-66.2%)
	`Retrofit`	3.6K	26	13 (-50.0%)	13 (-50.0%)	13 (-50.0%)
	Compiler Tools					
	`Jadx`	39.8K	493	865 (+75.5%)	132 (-73.2%)	124 (-74.8%)
	`WALA:Util`	19.5K	190	294 (+54.7%)	88 (-53.6%)	76 (-60.0%)
	Network Library					
	`Zuul`	15.2K	204	43 (-78.9%)	31 (-84.8%)	23 (-88.7%)
Uber	`T1`	35.7K	537	454 (-15.5%)	200 (-62.8%)	187 (-65.2%)
	`T2`	81.7K	1072	991 (-7.6%)	328 (-69.4%)	310 (-71.1%)
	`T3`	12.9K	229	134 (-41.5%)	46 (-79.9%)	31 (-86.5%)
	`T4`	20.1K	111	70 (-36.9%)	70 (-36.9%)	70 (-36.9%)
	`T5`	13.8K	222	192 (-13.5%)	126 (-43.2%)	126 (-43.2%)
	`T6`	3.4K	47	61 (+29.8%)	10 (-78.7%)	9 (-80.9%)
	`T7`	5.9K	35	28 (-20.0%)	21 (-40.0%)	19 (-45.7%)
	`T8`	14.8K	301	166 (-44.9%)	91 (-69.8%)	76 (-74.8%)

Using these two datasets, we seek to answer the following key research questions:

(1) Is NullAwayAnnotator effective in reducing the number of reported NullAway errors for these benchmarks?
(2) How does our technique compare with the strategy of applying all possible fixes, in terms of error reduction?
(3) What is the impact of the depth limit (see Section 3) on tool effectiveness, in terms of number of errors removed and running time?
(4) How much do our optimizations (Section 4) reduce running time over unoptimized?
(5) Is the output of NullAwayAnnotator an adequate basis to enable NullAway checking on previously-unannotated code in a production setting?

Section 8.2 addresses questions 1–3, Section 8.3 addresses question 4, and Section 8.4 addresses question 5.

Experiments for open-source benchmarks were performed on a desktop with an 11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz 8 core CPU and 32GB RAM running Ubuntu 20.04.5 LTS. Experiments at Uber were performed inside a Linux container on a shared AMD EPYC 2.45GHz machine, with 24 cores per socket and 2 sockets reporting 377 GB of RAM. We did not have dedicated access to this machine during our measurements, but it usually stayed at low utilization during experiments.

8.2 Error count reduction and depth bound

We first evaluated the effectiveness of NullAwayAnnotator at reducing the number of errors reported by NullAway. Data on this error reduction after running NullAwayAnnotator with various depth limits is shown in Table 1, with percentage changes in parentheses. The Initial column gives the number of NullAway errors before running inference. We show error reduction with depth limits 0, 1, and 5. Recall from Section 3 that a depth limit of 0 corresponds to eagerly inserting all possible fixes, ignoring the impact on error count.

Eager insertion of all possible fixes (depth limit 0) always yields more remaining errors than depth 1, and sometimes yields a higher number of errors than were reported on the original code! E.g., for `LibGdx`, the number of reported NullAway errors increases from 1549 to 2314 (49% more errors) at depth 0, and for `MpAndroid` the number of errors increases by 181%. These results show that our search strategy produces a final result with many fewer errors than performing eager `@Nullable` insertion.

To determine the impact of the depth limit, we ran NullAwayAnnotator with limits of 1–10 across all of our benchmarks. We observed that the final number of errors was never reduced further going beyond depth 5. Also, the running time at depth 5 was an average of 2.24X the depth 1 running time, a reasonable cost. So, on our benchmarks we concluded that depth 5 yielded the best tradeoff between performance and error reduction, and we used that depth limit for all subsequent experiments.

At depth 5, we saw an average reduction of 69.5% in the number of NullAway errors reported when compared to the initial code, ranging from 36.9% (for the `T4` Uber target) to 90.1% (for `SpringBoot`). The significant reduction of errors from running NullAwayAnnotator has multiple benefits: it enables more code to be checked immediately by NullAway via warning suppressions (see Section 8.4), and it reduces the effort required to eventually enroll all the code in NullAway checking.

8.3 Impact of optimizations

Next, we evaluated the performance impact of our graph-coloring optimizations (Section 4), and our results appear in Table 2. All configurations were run with a depth limit of 5 and a timeout of 8 hours. We show both the overall running time in minutes, and also the number of NullAway builds run in each configuration (nearly all execution time of NullAwayAnnotator is spent running NullAway builds, on average over 97% of running time for our open-source benchmarks). For our benchmarks, we observe enormous reductions in running times with the graph coloring optimization over unoptimized; the speedups range from 2.0X–17.8X, with an average of 6.1X. Further, for T1 and T2 we could not measure the speedup, as the unoptimized run could not complete within an 8-hour limit.

We observe a similarly-large reduction in number of builds required with optimizations enabled, showing that a significant number of the qualifiers evaluated by our search are in fact independent. Regarding the number of up-front builds required for client code (Section 5), we observed an average of 2.4 builds and a maximum of 8 across our benchmarks, small compared to the number of builds required during search. We conclude that the graph-coloring optimization provides an enormous benefit and is *essential* for making NullAwayAnnotator practical for larger programs.

Table 2: Running time and number of builds for unoptimized and optimized configurations.

	Benchmark Name	Time (Minutes)		Number of builds	
		Unoptimized	Optimized	Unoptimized	Optimized
Open Source Projects	`Conductor`	28.9	6.3 (4.6X)	351	115 (3.1X)
	`Mockito`	25.3	3.0 (8.4X)	383	71 (5.4X)
	`SpringBoot`	461.7	26.0 (17.8X)	1427	113 (12.6X)
	`LitiEngine`	59.0	5.1 (11.5X)	1122	98 (11.4X)
	`LibGdx`	339.2	27.7 (12.2X)	2320	195 (11.9X)
	`MPAndroid`	33.4	9.8 (3.4X)	402	119 (3.4X)
	`Glide`	28.9	5.1 (5.7X)	455	104 (4.4X)
	`EventBus`	1.5	0.5 (3.0X)	77	26 (3.0X)
	`Gson`	9.2	1.8 (5.1X)	181	66 (2.7X)
	`Eureka`	6.5	2.4 (2.7X)	198	72 (2.8X)
	`Retrofit`	0.6	0.3 (2.0X)	22	12 (1.8X)
	`Jadx`	68.7	10.5 (6.5X)	868	126 (6.9X)
	`WALA:Util`	76.11	12.37 (6.2X)	547	76 (7.2X)
	`Zuul`	25.3	2.3 (11.0X)	206	58 (3.6X)
Uber	`T1`	X	70.02 (-)	593+	77 (-)
	`T2`	X	114.02 (-)	706+	89 (-)
	`T3`	183.19	44.42 (4.1X)	404	91 (4.4X)
	`T4`	33.02	13.74 (2.4X)	53	18 (2.9X)
	`T5`	292.06	61.28 (4.8X)	559	110 (5.1X)
	`T6`	24.01	9.75 (2.5X)	59	19 (3.1X)
	`T7`	31.64	12.48 (2.5X)	72	22 (3.3X)
	`T8`	374.44	71.69 (5.2X)	842	154 (5.5X)

Table 3: Number of annotations injected by NullAwayAnnotator.

	Benchmark Name	Depth 5			Suppress-only
		# @Nullable	# Suppression	% unchecked	% unchecked
Open Source Projects	`Conductor`	319	31	5.95%	22.6%
	`Mockito`	322	31	3.6%	14.82%
	`SpringBoot`	1331	81	1.99%	12.03%
	`LitiEngine`	993	158	5.85%	12.08%
	`LibGdx`	1426	459	4.77%	10.04%
	`MPAndroid`	253	66	2.49%	6.27%
	`Glide`	370	99	5.68%	9.57%
	`EventBus`	71	9	5.44%	14.97%
	`Gson`	202	21	5.2%	18.42%
	`Eureka`	138	30	6.04%	11.71%
	`Retrofit`	27	10	16.79%	17.1%
	`Jadx`	509	139	4.98%	12.18%
	`WALA:Util`	165	67	4.27%	7.66%
	`Zuul`	164	27	5.13%	9.15%
Uber	`T1`	501	177	8.54%	13.16%
	`T2`	863	290	8.59%	15.09%
	`T3`	320	28	3.89%	19.55%
	`T4`	322	46	10.15%	13.64%
	`T5`	231	103	14.61%	20.67%
	`T6`	35	10	2.12%	10.18%
	`T7`	88	19	4.42%	6.52%
	`T8`	656	81	8.64%	7.66%

8.4 Tool output and real-world usage

Here, we characterize the inferred annotations discovered by NullAwayAnnotator and describe its real-world usage thus far. Table 3 gives data on the final solutions found by NullAwayAnnotator for our benchmarks. For depth 5, we show the number of `@Nullable` annotations inferred, the number of annotations inserted to suppress remaining warnings, and the percentage of code that remains unchecked by NullAway due to these suppressions. We also show the percentage of unchecked code for a baseline configuration, in which all the errors initially reported by NullAway are suppressed (without running inference). Due to a tool bug, for one benchmark (T5) we added one suppression annotation manually.

The number of `@Nullable` annotations inserted by NullAwayAnnotator is significant (up to 1,426), reflecting the large amount of manual work otherwise required to adopt NullAway. After suppressions were inserted at depth 5, the percentage of unchecked code ranged from 1.99% to 16.79%, with an average of 6.32%. In comparison to the baseline of suppressing with no inference, the percentage of unchecked code decreased by an average factor of 2.54X (0.89X to 6.05X). For the one target where the amount of unchecked code increased with inference (the Uber T8 module), the baseline suppresses many uninitialized field warnings at the field declarations, while inference makes the fields `@Nullable` and adds suppressions on certain methods using the fields. Even though inference adds many fewer suppressions (81 vs. 362 for the baseline), due to the placement of these suppressions on methods rather than fields, the amount of unchecked code becomes higher. Table 1 shows that inference still dramatically reduces the number of NullAway errors for this benchmark.

Overall, the data show that our inference allows for a much greater amount of existing code to be immediately checked by NullAway, yielding greater safety for future code modifications with no manual effort.

Initializers We manually inspected all injected `@Initializer` annotations, and all were correct except for two in `LibGdx`. For the bad cases, the method was in fact a setter not involved in initialization, though it assigned values to multiple fields. In the future, we believe we can make `@Initializer` inference more accurate by leveraging method naming patterns (e.g., including methods with names like `init` while excluding methods whose names start with `set`).

Deployment NullAwayAnnotator has been deployed at Uber for self-serve use by developers. Thus far, it has been used to enroll 160 modules in NullAway checking, consisting of roughly 1.365 million lines of Java code. Running NullAwayAnnotator on production modules led to many bug fixes and improvements in the tool itself, e.g., handling of code generation by annotation processors. The main lesson from our experience thus far is that NullAwayAnnotator changes should as much as possible be scoped to a single module and be semantics-preserving, to avoid long code review cycles (see Section 5). A key issue with extended code review is that the target and client code keeps evolving, and keeping the inferred changes consistent with the evolving code requires significant manual effort.

We also used NullAwayAnnotator to enable NullAway checking for the WALA:Util open-source module, maintained by one of the paper authors. Here, we again found NullAwayAnnotator's support for analyzing client code to be useful, as WALA:Util is used by many other modules in the project, and we wanted the annotations to capture that usage behavior. In studying the final output, we found several places where NullAwayAnnotator inferred a `@Nullable` annotation for a field or method parameter, but the author would have preferred to refactor the code to make that location `@NonNull`. Automatically performing such code refactorings and repairs is out of scope for NullAwayAnnotator, but is a fruitful avenue for further research.

The `@Nullable` annotations discussed above could be considered "false positives" since they do not match the annotations a developer would have written by hand; we expect that similar cases may have occurred in our other benchmarks. We have found that in such cases, introducing the desired developer annotations usually also requires modifying executable program code. Any change that modifies executable code require much deeper review than the changes generated by NullAwayAnnotator, which are semantics-preserving. For effective deployment, we believe future repair approaches would be best deployed in combination with NullAwayAnnotator; NullAwayAnnotator's changes would enable immediate NullAway checking with no review required, and subsequent automated repair patches could be reviewed and incorporated gradually.

8.5 Threats to Validity and Limitations

The main threat to the external validity of our evaluation is our choice of benchmarks. We strove to choose a diverse set of benchmarks in a principled manner (see Section 8.1). Still, it is possible that on less popular open-source benchmarks, or on benchmarks using build systems besides Gradle, NullAwayAnnotator will be less effective. And, it is possible that NullAwayAnnotator is particularly effective for code written in the style used at Uber, but that it will be less effective for other proprietary code. Regarding internal validity, our results may be impacted by implementation bugs in NullAwayAnnotator. To combat this issue, NullAwayAnnotator has an extensive suite of unit and integration tests. Further, for all benchmarks, we verified that the final result of NullAwayAnnotator was exactly the same with and without optimizations enabled. We have also manually vetted the output of NullAwayAnnotator on several Uber targets.

A limitation of NullAwayAnnotator is that it does not support inference of all annotations supported by NullAway. We do not yet support inference of `@Contract`, `@RequiresNonNull`, and `@EnsuresNonNull` annotations, used to express pre- and post-conditions (e.g., that some field of a parameter must be `@NonNull` at method entry). We focused on inference of `@Nullable` annotations initially, since it provides a large benefit on its own, but we plan to support inference of pre- and post-condition annotations in the future. Other nullness checkers [28, 31] support writing type qualifiers on generic type arguments (e.g., `List<@Nullable String>`), but NullAway does not. NullAwayAnnotator does not currently support inference of such qualifiers, but we are working to extend it with such support.

9 RELATED WORK

There is a wide and rich literature on classical type inference (or type reconstruction) [32, Chapter 22], which focuses on discovering whether there exists a (complete) typing for an unannotated program in a given type system. Our problem differs from the classical case as we nearly always target programs where no such typing exists, and our goal is to find a maximal set of useful type qualifiers for such programs. As such, we focus our related work discussion on techniques more closely related to our target scenario, and do not discuss type inference work more broadly.

Checker Framework Inference [7] uses constraint-based analysis to infer types, and the approach has been applied successfully to a type systems for measurement units [38]. This approach solves the constraints using MaxSAT, which could be adapted to output a partial typing for the program when the constraints are unsatisfiable. Previous approaches to improved error explanation for type inference [23, 29, 39] and migration of dynamically-typed programs to use gradual types [6, 25, 30] are also based on constraints, and could be similarly adapted. However, as noted in Section 1, we require reuse of an existing checker implementation for our scenario, and cannot re-implement the checker using constraints.

The Checker Framework includes whole-program inference (WPI) functionality [20] that works with unmodified pluggable type system implementations. The technique works by inserting the most specific type qualifier compatible with all expressions written into an entity (a field, parameter, or return value), running to a fixed point. WPI is integrated into the Checker Framework, and hence can infer many annotations from a single run. Our technique treats the checker as a black box, necessitating the optimizations of Section 4 for better performance. Our less-coupled approach makes it potentially easier to combine our technique with other checker implementations, and we plan to explore integrations with the Checker Framework. WPI is *not* guided by minimizing the final number of errors reported; its strategy resembles that of our eager insertion of qualifiers, which Section 8 showed can *increase* the final number of errors. Also, our technique aims to generate annotations a developer would accept into their source code. WPI may generate many annotations unrelated to any reported error, which developers are unlikely to incorporate (to minimize clutter).

The Daikon dynamic invariant detector [11] can infer `@Nullable` annotations from dynamic behaviors [8]. This approach infers a `@Nullable` qualifier only for locations observed to be `null` at runtime, a guarantee that any static approach cannot provide. As with any dynamic approach, it requires that the target code to be executable by the tool and that some set of suitable inputs is available. Static approaches like ours are complementary, as they need not be able to execute the program and can account for all possible code behaviors.

Cascade [36] is an interactive type qualifier inference tool that involves programmers in the inference process. Cascade also targets programs where code changes are likely to be required to make the code type check. NullAwayAnnotator aims to automate more of the qualifier inference process than Cascade. The two approaches are complementary; after adopting the initial annotations proposed by NullAwayAnnotator, a developer could use a Cascade-like tool to aid in gradually fixing the remaining errors.

Recent work has applied modern machine learning techniques to type inference [16, 34] and to program repair whose fixes may include type qualifier insertion [3]. We have not yet pursued such techniques due to the amount of training data required; we are not aware of a publicly-available data set showing how `@Nullable` qualifiers are inserted to address type errors. In the future we plan to investigate generation of training data [5] to further enable a learning-based approach. Note that TypeWriter [34] also uses black-box executions of an extant type checker to evaluate candidate types; we believe our graph-coloring optimization could be used to reduce the number of type checker runs required by their technique.

10 CONCLUSIONS

We have presented a novel approach to inference of nullability qualifiers for Java programs, to enable applying nullness type checkers to extant code bases. In contrast to many other techniques, our approach treats the type checker as a black-box oracle and does not require re-implementation of its logic. We defined an effective search strategy for discovering a good set of qualifiers to insert, and presented optimizations to dramatically speed up the search. We implemented our approach in an open-source tool NullAwayAnnotator and evaluated it on both open-source and commercial code bases. Our evaluation showed that NullAwayAnnotator scaled well, and that the inferred annotations significantly reduced the number of errors reported, enabling NullAway checking for more existing code. NullAwayAnnotator has already been used to enable NullAway checking for 160 production modules at Uber. In future work, we plan to generalize our approach to other static type and type-qualifier systems.

11 DATA AVAILABILITY

NullAwayAnnotator is open source and available at https://github.com/ucr-riple/NullAwayAnnotator. Further, we have made an artifact available at https://zenodo.org/record/8271236 containing the code for NullAwayAnnotator and scripts to run it on our open-source benchmarks.

ACKNOWLEDGMENTS

We would like to thank all the Uber developers who contributed to this work by using NullAwayAnnotator to annotate their code and providing feedback on the tool, with special thanks to Ameya Ketkar, Sonal Mahajan, Yuxin Wang, Raj Barik, and Swati Gupta for their help coordinating the initial code enrollments.

This research was supported in part by the National Science Foundation under grants CCF-2007024 and CCF-2312263, a gift from Oracle Labs, and a Google Research Award. This research was partially sponsored by the OUSD(R&E)/RT&L and was accomplished under Cooperative Agreement Number W911NF-20-2-0267. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL and OUSD(R&E)/RT&L or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] ActivityLifecycle 2022. The Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed: 2022-04-12.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (oct 2019), 27 pages. https://doi.org/10.1145/3360585
[4] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-based Null Safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* https://doi.org/10.1145/3338906.3338919
[5] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin T. Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. http://proceedings.mlr.press/v139/berabi21a.html
[6] John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *Proc. ACM Program. Lang.* 2, POPL, Article 15 (2018), 29 pages. https://doi.org/10.1145/3158103
[7] CFInference 2022. Checker Framework Inference. https://github.com/opprop/checker-framework-inference. Accessed: 2022-04-02.
[8] DaikonNullable 2022. Daikon AnnotateNullable support. http://plse.cs.washington.edu/daikon/download/doc/daikon.html#AnnotateNullable. Accessed: 2022-04-12.
[9] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. 2011. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Hawaii, USA, 681–690. https://doi.org/10.1145/1985793.1985889
[10] Eradicate 2022. Infer : Eradicate. https://fbinfer.com/docs/checker-eradicate Accessed: 2022-04-07.
[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015
[12] ErrorProne 2022. Error Prone. http://errorprone.info/ Accessed: 2022-04-07.
[13] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. 192–203. https://doi.org/10.1145/301618.301665
[14] Gradle 2022. Gradle Build Tool. https://gradle.org. Accessed: 2022-04-03.
[15] gRPC 2022. gRPC. https://grpc.io. Accessed: 2022-04-07.
[16] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051
[17] David Hovemeyer, Jaime Spacco, and William Pugh. 2005. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Lisbon, Portugal) *(PASTE '05)*. Association for Computing Machinery, New York, NY, USA, 13–19. https://doi.org/10.1145/1108792.1108798
[18] Thore Husfeldt. 2015. Graph colouring algorithms. https://doi.org/10.48550/ARXIV.1505.05825 arXiv pre-print 1505.05825.
[19] JavaParser 2022. JavaParser. https://javaparser.org. Accessed: 2022-04-02.
[20] Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, and Michael D. Ernst. 2023. Pluggable type inference for free. In *ASE 2023: Proceedings of the 38th Annual International Conference on Automated Software Engineering.*
[21] Kotlin 2022. Kotlin Programming Language. https://kotlinlang.org/. Accessed: 2022-04-07.
[22] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. 2008. Verifying Dereference Safety via Expanding-Scope Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/1390630.1390657
[23] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. *SIGPLAN Not.* 51, 10 (oct 2016), 781–799. https://doi.org/10.1145/3022671.2983994
[24] Ravichandhran Madhavan and Raghavan Komondoor. 2011. Null dereference verification via over-approximated weakest pre-conditions analysis. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 1033–1052. https://doi.org/10.1145/2048066.2048144
[25] Zeina Migeed and Jens Palsberg. 2020. What is Decidable about Gradual Types? *Proc. ACM Program. Lang.* 4, POPL, Article 29 (2020), 29 pages. https://doi.org/10.1145/3371097
[26] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (Eds.). ACM, 284–292. https://doi.org/10.1145/1062455.1062514
[27] NAHandlers 2022. NullAway handler implementations. https://github.com/uber/NullAway/tree/master/nullaway/src/main/java/com/uber/nullaway/handlers. Accessed: 2022-04-07.
[28] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Seattle, WA, USA, 201–212. https://doi.org/10.1145/1390630.1390656
[29] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 525–542. https://doi.org/10.1145/2660193.2660230
[30] Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-Based Gradual Type Migration. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 111 (oct 2021), 27 pages. https://doi.org/10.1145/3485488
[31] Artem Pianykh, Ilya Zorin, and Dmitry Lyubarskiy. 2023. Retrofitting null-safety onto Java at Meta. https://engineering.fb.com/2022/11/22/developer-tools/meta-java-nullsafe/ Accessed: 2023-02-01.
[32] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.
[33] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (jun 2016), 78–87. https://doi.org/10.1145/2854146
[34] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3368089.3409715
[35] Swift 2022. Swift Programming Language. https://swift.org/. Accessed: 2022-04-07.
[36] Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. 2015. Cascade: A Universal Programmer-Assisted Type Qualifier Inference Tool. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 234–245.
[37] WALA 2023. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.
[38] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. 2020. Precise inference of expressive units of measurement types. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 142:1–142:28. https://doi.org/10.1145/3428210
[39] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2017. SHErrLoc: A Static Holistic Error Locator. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 18 (aug 2017), 47 pages. https://doi.org/10.1145/3121137

Received 2023-02-02; accepted 2023-07-27