NullAway: Practical Type-Based Null Safety for Java

Anonymous Author(s)

ABSTRACT

`NullPointerException`s (NPEs) are a key source of crashes in modern Java programs. Previous work has shown how such errors can be prevented at compile time via code annotations and pluggable type checking. However, such systems have been difficult to deploy on large-scale software projects, due to significant build-time overhead and / or a high annotation burden. This paper presents NullAway, a new type-based null safety checker for Java that overcomes these issues. NullAway has been carefully engineered for low overhead, so it can run as part of every build. Further, NullAway reduces annotation burden through targeted unsound assumptions, aiming for *no false negatives in practice* on checked code. Our experimental evaluation showed that NullAway has significantly lower build-time overhead (1.15×) than comparable tools (2.7-5.9×). Further, on a corpus of production crash data for widely-used Android apps built with NullAway, remaining NPEs were almost always due to unchecked third-party libraries (64%), deliberate error suppressions (17%), or reflection and other forms of post-checking code modification (17%), *never* due to NullAway's unsound assumptions for checked code.

ACM Reference Format:
Anonymous Author(s). 2019. NullAway: Practical Type-Based Null Safety for Java. In *Proceedings of The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

`NullPointerException`s (NPEs), caused by a dereference of `null`, are a frequent cause of crashes in modern Java applications. Such crashes are nearly always troublesome, but they are particularly problematic in mobile applications. Unlike server-side code, where a bug fix can be deployed to all users quickly, getting a fixed mobile app to users' devices can take days to weeks, depending on the app store release process and how often users install updates. Due to the severity of null-dereference errors, recent mainstream languages like Swift [11] and Kotlin [8] enforce null safety as part of type checking during compilation.

ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Previous work has added type-based null safety to Java, via code annotations and additional type checking [6, 21, 28]. With this approach, developers use `@Nullable` and `@NonNull` code annotations to indicate whether entities like fields, parameters, and return values may or may not be `null`. Given these annotations, a tool checks that the code is null safe, by ensuring, e.g., that `@Nullable` expressions are never dereferenced and that `null` is never assigned to a `@NonNull` variable. Previous work has shown this approach to be an effective way to prevent NPEs [21, 28].

Despite their effectiveness, previous type-based null safety tools for Java suffered from two key drawbacks. First, the build-time overhead of such tools is quite high. Our experimental evaluation showed the two best-known tools to have average overheads of 2.7× and 5.9× respectively (see Section 8) compared to regular compilation. For a seamless development experience, a null safety tool should run every time the code is compiled, but previous tool overheads are too high to achieve this workflow without excessive impact on developer productivity. Second, some previous tools prioritize soundness, i.e., providing a strong guarantee that any type-safe program will be free of NPEs. While this guarantee is appealing in principle, it can lead to significant additional annotation burden for developers, limiting tool adoption.

To address these drawbacks, we have developed NullAway, a new tool for type-based null safety for Java. NullAway runs as a plugin to the Error Prone framework [16], which provides a simple API for extending the Java compiler with additional checks. The core of NullAway includes similar features to previous type-based null safety tools, including defaults that reduce the annotation burden and flow-sensitive type inference and refinement [28]. NullAway also includes additional features to reduce false positives, including support for basic pre- and post-conditions and special support for stream-based APIs (see Section 6).

NullAway is carefully engineered and regularly profiled to ensure low build-time overhead. We built NullAway at Uber Technologies Inc. (Uber), and have run it as part of all our Android builds (both on continuous integration servers and developer laptops) for nearly two years. At Uber, NullAway replaced another tool which, due to performance limitations, ran only at code review time. Running NullAway on all builds enabled much faster feedback to developers.

In terms of soundness, NullAway aims to have *no false negatives in practice for code that it checks*, while reducing the annotation burden wherever possible. NullAway's checks to ensure `@NonNull` fields are properly initialized (Section 3) are unsound, but also require far fewer annotations than a previous sound checker [1, §3.8]. Similarly, NullAway unsoundly assumes that methods are pure and side-effect free (Section 4). In both cases, we have validated this checking based on crash data from the field for Uber's Android apps,

showing that neither source of unsoundness seems to lead to real-world NPEs for our code.

For usability, NullAway uses an *optimistic* handling of calls into unchecked code, though such handling can lead to uncaught issues. Modern Java projects often depend on numerous third-party libraries, many of which do not yet contain nullability annotations on their public APIs. Full soundness requires a *pessimistic* modeling of such libraries, with worst-case assumptions about their nullness-related behavior. But, pessimistic assumptions lead to a large number of false positive warnings, in our experience making the tool unusable on a large code base. Instead, NullAway treats calls into unchecked code *optimistically*: it is assumed that such methods can always handle `null` parameters and will never return `null`.[1] Additionally, NullAway includes mechanisms for custom library modeling and leveraging nullability annotations when they are present. Overall, NullAway's handling of unchecked code is practical for large code bases while providing mechanisms for additional safety where needed.

Beyond Uber, NullAway is freely available and open source. It has been adopted by a number of other companies and open-source projects, further validating its usefulness and design decisions.

We performed an extensive experimental evaluation of NullAway, on 18 open-source projects totaling $\sim 164K$ lines of code and on $\sim 3.3M$ lines of production code from widely-used Android apps developed at Uber. We observed that NullAway introduced an average of 15% overhead to build times on the former (22% on the later), significantly lower than previous tools. Further, a study of one month of crash data from Uber showed that NPEs were uncommon, and that nearly all remaining NPEs were due to interactions with unchecked code, suppression of NullAway warnings, or post-checking code modification. *None* of the NPEs were due to NullAway's unsound assumptions for checked code. Finally, the evaluation confirmed that removing these unsound assumptions leads to significantly more warnings for developers.

Contributions This paper makes the following contributions:

- We describe the design of NullAway's type system, tuned over many months to achieve no false negatives in practice for checked code with a reasonable annotation burden. NullAway includes carefully-designed initialization checking (Section 3), an optimistic treatment of function purity and side effects (Section 4), and a highly-configurable system for determining how to treat unchecked code (Section 5). Our evaluation showed that a checker without these unsound assumptions emitted many false positive warnings (Section 8.4).
- We present experiments showing that NullAway's build-time overhead is dramatically lower than alternative systems, enabling NPE checking on every build (Section 8.2).
- We analyze production crash data for a large code base built with NullAway and show that on this data set, NullAway achieved its goal of no false negatives for checked code, as remaining NPEs were primarily caused by third-party libraries and warning suppressions (Section 8.3).

[1] Optimistic handling is also used for overriding methods from unlisted packages; see Section 5.

2 OVERVIEW

In this section we give a brief overview of type-based nullability checking as implemented in NullAway. The core ideas of preventing NPEs via pluggable types are well known; see elsewhere [1, 21, 28] for further background.

With type-based null checking, a type's nullability is expressed via additional qualifiers, written as annotations in Java. The `@NonNull` qualifier describes a type that excludes `null`, whereas `@Nullable` indicates the type includes `null`. Given these additional type qualifiers, type checking ensures the following two key properties:

(1) No expression of `@Nullable` type is ever assigned to a location of `@NonNull` type.
(2) No expression of `@Nullable` type is ever dereferenced.

Together, these properties ensure a program is free of NPEs, assuming objects have been properly initialized. (We defer discussion of initialization checking to Section 3.)

Consider the following simple example:

```
1 static void log(@NonNull Object x) {
2     System.out.println(x.toString());
3 }
4 static void foo() { log(null); }
```

Here, the parameter of `log` is `@NonNull`, so the call `log(null);` will yield a type error, as it violates property 1.[2] The developer could address this issue by changing the annotations on `log`'s parameter `x` to be `@Nullable`. But, `x` is dereferenced at the call `x.toString()`, which would yield another type error due to violating property 2.

The developer can make the code type check as follows:

```
5 static void log(@Nullable Object x) {
6   if (x != null) { System.out.println(x.toString()); }

7 }
```

The type checker proves this code safe via *flow-sensitive type refinement* (to be discussed further at the end of this section); the checker interprets the null check and refines `x`'s nullness type to be `@NonNull` within the if-body, making the `toString()` call legal.

Types qualified with nullability annotations form a subtyping relationship, where `@NonNull C` is a subtype of `@Nullable C` for any class `C` [28, Fig. 3]. Hence, property 1 simply ensures assignment compatibility according to subtyping.

Override Checking Beyond checking assignments, the checker must also ensure that method overrides respect subtyping. Consider the following example:

[2] For type checking, "assignments" include parameter passing and returns at method calls.

```
8  class Super {
9    @NonNull Object getObj() { return new Object(); }
10 }
11 class Sub extends Super {
12   @Nullable getObj() { return null; }
13 }
14 class Main {
15   void caller() {
16     Super x = new Sub();
17     x.getObj().toString(); // NullPointerException!
18   }
19 }
```

Since `x` has declared type `Super`, the declared target of `x.getObj()` on line 17 is `Super.getObj`. This method has a `@NonNull` return type, making the `toString()` call legal. However, this example crashes with an NPE, since overriding method `Sub.getObj` has `@Nullable` return type. To close this loophole, the checker must ensure that a method with `@NonNull` return type is not overridden by a method with `@Nullable` return type. Similarly, it must check that if a method has a `@Nullable` parameter, an overriding method does not make the parameter `@NonNull`.

Defaults Annotating every field, parameter, and return value in a large code base would require a huge effort. NullAway uses the non-null-except-locals (NNEL) default from the Checker Framework [28] to reduce the annotation burden. Any unannotated parameter, field, or return value is treated as `@NonNull`, whereas the types of local variables are inferred (see below). Beyond reducing annotation effort, this default makes code more readable (by reducing annotation clutter) and nudges the developer away from using `null` values, making the code safer.

Flow-Sensitive Type Inference / Refinement As in previous work, NullAway automatically infers types for local variables in a flow-sensitive manner. Beyond inspecting assignments, null checks in conditionals are interpreted to compute refined (path-sensitive) types where the condition holds. E.g., at line 6 of the previous `log` example, the type of `x` is refined to `@NonNull` inside the if-body, based on the null check. NullAway uses an access-path-based abstract domain [20] to also track nullability of sequences of field accesses and method calls. Section 4 describes how NullAway's assumptions around side effects and purity interact with type inference.

Other Tools Throughout this paper we discuss two other type-based null checking tools for Java: the Nullness Checker from the Checker Framework [21, 28], which we refer to as CFNullness for brevity, and Eradicate, available with Facebook Infer [6]. Subsequent sections will detail how the checks performed by NullAway and its overheads compare with CFNullness and Eradicate.

3 INITIALIZATION CHECKING

Beyond the checks shown in Section 2, to fully prevent NPEs a nullness type checker must ensure that objects are properly initialized. Sound type systems for checking object initialization have been a subject of much previous research [18, 22, 23, 29, 30]. In this section, we present NullAway's approach to initialization checking. Though unsound, our technique has a low annotation burden and has caught nearly all initialization errors in our experience at Uber.

```
20 class InitExample {
21   @NonNull Object f, g, h, k;
22   InitExample() {
23     this.f = new Object();
24     this.g.toString(); // use before init
25     helper();
26   }
27   private void helper() {
28     this.g = new Object();
29     this.h.toString(); // use before init
30   }
31   @Initializer
32   public void init() {
33     this.h = this.f;
34     if (cond()) { this.k = new Object(); }
35   }
36 }
```

Figure 1: An example (with errors) to illustrate initialization checking.

Figure 1 gives a code example we will use to illustrate our initialization checking. We first describe how NullAway checks that `@NonNull` fields are initialized (Section 3.1), then discuss checking for uses before initialization (Section 3.2), and then compare with CFNullness and Eradicate (Section 3.3).

3.1 Field Initialization

Initialization phase Any `@NonNull` instance field must be assigned a non-null value by the end of the object's initialization phase.[3] We consider an object's initialization phase to encompass execution of a constructor, possibly followed by *initializer methods*. Initializer methods (or, simply, initializers) are methods invoked at the beginning of an object's lifecycle but after its constructor, e.g., overrides of `onCreate()` in Android `Activity` subclasses [14]. Field initialization may occur directly in constructors and initializers, or in invoked helper methods.

In Figure 1, the `InitExample` class has four `@NonNull` fields, declared on line 21. NullAway treats the `init()` method (lines 32–35) as an initializer, due to the `@Initializer` annotation. For a method annotated `@Initializer`, NullAway *assumes* (without checking) that client code will *always* invoke the method before other (non-initializer) methods in the class. Note that the `InitExample` constructor invokes `helper()` at line 25 to perform some initialization.

[3] For space, we elide discussion of NullAway's handling of static field initialization; the techniques are roughly analogous to those for instance fields.

Checks Given a class C with constructors, initializers, and initializer blocks [13, §8.6], for each `@NonNull` field f of C, NullAway treats f as properly initialized if at least one of four conditions hold:

(1) f is initialized directly at its declaration; or
(2) f is initialized in an initializer block; or
(3) C has at least one constructor, and *all* constructors initialize f; or
(4) *some* initializer in C initializes f.

For a method, constructor, or initializer block m to initialize a field f, f must *always* be assigned a non-null value by the end of m. This property can be determined using the same analysis used for flow-sensitive type inference (see Section 2), by checking if the inferred type of `this.f` is `@NonNull` at the end of m. NullAway also allows for initialization to occur in a method that is *always invoked* by m. NullAway determines if m always invokes a method n with two simple checks: (1) the call to n must be a top-level statement in m (not nested within a conditional or other block), and (2) n must be `private` or `final`, to prevent overriding in subclasses.[4]

Returning to Figure 1, NullAway reasons about field initialization as follows:

- `f` is properly initialized due to the assignment at line 23.
- `g` is properly initialized, since the constructor always invokes `helper()` (line 25), which assigns `g` (line 28).
- `h` is properly initialized, since `@Initializer` method `init()` assigns `h` (line 33).
- line 34 only initializes `k` conditionally. So, NullAway reports an error that `k` is not properly initialized.

3.2 Use before Initialization

Within the initialization phase, a further check is required to ensure that `@NonNull` fields are not used before they are initialized. Two such bad uses exist in Figure 1: the read of `this.g` at line 24 and `this.h` at line 29. NullAway performs a partial check for these bad uses. Within constructors and initializers, NullAway checks at any field use that the field is definitely initialized before the use. This check again leverages the same analysis used for flow-sensitive type inference. NullAway must also account for fields that have been initialized before the analyzed method. For example, the read of `this.f` at line 33 of Figure 1 is safe, since `f` is initialized in the constructor, which runs earlier. Similarly NullAway accounts for fields initialized in always-invoked methods before a read.

NullAway's check is partial since it does not check field reads in methods invoked by constructors or initializers or guard against other leaking of the `this` reference during initialization. So, while NullAway reports a use-before-init error at line 24 of Figure 1, it does not report an error for the uninitialized read at line 29. While handling certain cases like reads in always-invoked methods would be straightforward, detecting all possible uninitialized reads would be non-trivial and add significant complexity to NullAway. Uninitialized reads beyond those detected by NullAway seem to be rare, so we have not yet added further checking.

[4] NullAway does not attempt to identify methods that are always invoked from a constructor or initializer through a chain of invocations more than one level deep; this has not led to false positive warnings in practice.

```
37 class FooHolder {
38   @Nullable Object foo;
39   @Nullable public Object getFoo() { return this.foo; }
40   public void setFoo(@Nullable Object foo) {
41     this.foo = foo;
42   }
43   @Nullable public Object getFooOrNull() {
44     return randInt() > 10 ? null : this.foo;
45   }
46 }
```

Figure 2: An example to illustrate side effect and purity handling in NullAway.

3.3 Discussion

In contrast to NullAway, CFNullness aims for sound initialization checking. The CFNullness initialization checking system [1, §3.8] (an extension of the Summers and Müller type system [30]) prohibits invoking any method on a partially-initialized object without additional developer annotations. E.g., CFNullness prohibits the call at line 25, since `helper()` is not annotated as being able to operate during initialization. It also lacks support for a direct analogue of the `@Initializer` annotation. As we shall show in Section 8 this strict checking leads to a number of additional false warnings. NullAway's checking is unsound, but it seems to catch most initialization errors in practice with a much lower annotation burden.

NullAway's initialization checking was inspired by the checking performed in Eradicate, which also supports the `@Initializer` annotation. Compared with Eradicate, there are two main differences in how NullAway checks for initialization. First, NullAway only considers initialization from callees that are always invoked (see Section 3.1). In contrast, Eradicate considers initialization performed in all (private or final) constructor callees, even those invoked conditionally, which is less sound. E.g., if line 25 were written as `if (cond()) helper();`, Eradicate would still treat fields assigned in `helper` as initialized. Second, Eradicate does not have any checking for use before initialization (Section 3.2).

Note that usage of `@Initializer` can be dangerous, as NullAway does not check that such methods are invoked before others. In the Uber code base most usage of `@Initializer` is via overriding of well-known framework methods like `Activity.onCreate`. When developers introduce new usage of `@Initializer`, our code review system automatically adds a comment to warn about the risks.

4 SIDE EFFECTS AND PURITY

NullAway reduces warnings (unsoundly) by assuming all methods are both side-effect-free and pure. Figure 2 gives

a simple example of a class `FooHolder` that has a `foo` field with a getter and setter. NullAway's flow-sensitive type inference assumes method calls are side-effect-free, so it will (erroneously) not report a warning on this code:

```
47  FooHolder f = ...;
48  if (f.foo != null) {
49    f.setFoo(null);
50    f.foo.toString(); // NPE!
51  }
```

NullAway ignores the effect of the `setFoo()` call and assumes `f.foo` remains non-null at line 50, based on the null check at line 48. In practice, we have not observed any NPEs in the field due to this type of side effect. In the Uber code base most data-holding classes are immutable, precluding this type of error. Also, for this type of code usually the dereference immediately follows the null check (with no intervening code), a safe pattern even with mutable types.

Additionally, NullAway assumes all methods are pure, in order to refine nullability of "getter" return values during type inference. The following code is NPE-free:[5]

```
52  FooHolder f = ...;
53  if (f.getFoo() != null) {
54    f.getFoo().toString();
55  }
```

NullAway refines the nullability of `getFoo()`'s return to be `@NonNull` under the condition, to avoid emitting a warning. However, if instead of calling `getFoo()` the code called `getFooOrNull()` (see line 43 in Figure 2), an NPE would be possible, and NullAway would not report it. The issue is that `getFooOrNull()` is *impure*: it can return different values across multiple calls with the same arguments. We have not observed impurity to cause soundness issues for NullAway in practice. Note that NullAway limits type refinement of return nullability to those methods with no arguments besides the receiver (like getters); it may be that such methods are particularly likely to be pure.

By default, CFNullness soundly assumes that methods may be impure and have side effects. While this catches more bugs, on the Uber code base this would lead to a huge number of false warnings. CFNullness has an option to assume methods are side-effect free, but no option as of yet to assume purity.

5 HANDLING UNANNOTATED CODE

This section details how NullAway handles interactions with unannotated, unchecked code, typically written by a third-party. Since modern Java programs often use many third-party libraries without nullability annotations, these interactions arise frequently in real-world code. By default, NullAway uses an unsound, *optimistic* handling of interactions with unannotated code, sacrificing some safety to enhance tool usability.

[5]This ignores multithreading, for which NullAway is unsound; CFNullness is also unsound for multithreading by default.

Assume that code in a program has been partitioned into *checked* code, which has proper nullability annotations checked by NullAway, and *unannotated* code, which is lacking annotations and has not been checked. (We shall detail how this partition is computed shortly.) By default, NullAway treats any interaction between the checked and unannotated code *optimistically*, i.e., with the assumption that no errors will arise from the interaction. In particular, this means:

- When checking a call to an unannotated method m, NullAway assumes that m's parameters are `@Nullable` and that m's return is `@NonNull`.
- When checking an override of an unannotated method m (see discussion of override checking in Section 2), NullAway assumes that m's parameters are `@NonNull` and that m's return is `@Nullable`.

These assumptions are maximally permissive and ensure that no errors will be reported for interactions with unannotated code, a clearly unsound treatment.

Alternatives to optimistic handling of unannotated code yield too many false positive warnings to be usable. No handling of third-party code can truly prevent all NPEs, as there may be NPE bugs within the third-party code independent of what exact values are passed to API methods. A maximally-safe handling of interactions with third-party code would be *pessimistic*, making the exact opposite assumptions made in optimistic checking (e.g., all unannotated return values would be treated as `@Nullable`). This handling would lead to a huge number of false warnings, as the assumptions are much too conservative. By default, CFNullness handles third-party libraries with the same defaults as first-party code: any parameter or return missing an annotation is assumed to be `@NonNull`. As we shall show in Section 8, these assumptions also lead to a large number of false warnings.

NullAway has a highly-configurable system for specifying which code is unannotated and how optimistically it is handled. At the highest level, annotated and unannotated code is partitioned based on its Java package, *not* whether the code is first-party or third-party. This system provides a high degree of flexibility when adopting NullAway; source packages can be treated as unannotated for gradual adoption, while third-party packages can be treated as annotated if they have proper annotations present.

Figure 3 presents a flow chart showing how NullAway determines the nullability of the first argument to a hypothetical method `a.b.C.foo(Object o)` (represented by the missing annotation placeholder `?`). The first four steps seek to determine whether the code is annotated or unannotated. The method is treated as annotated if (1) the package name matches the `AnnotatedPackages` regex, (2) it does not match the `UnannotatedSubPackages` regex, (3) the class name is not blacklisted in `UnannotatedClasses`, and (4) the class is not annotated as `@Generated` with the option `TreatGeneratedAsUnannotated` set. In this case, the nullability of an argument is assumed to be `@NonNull`.

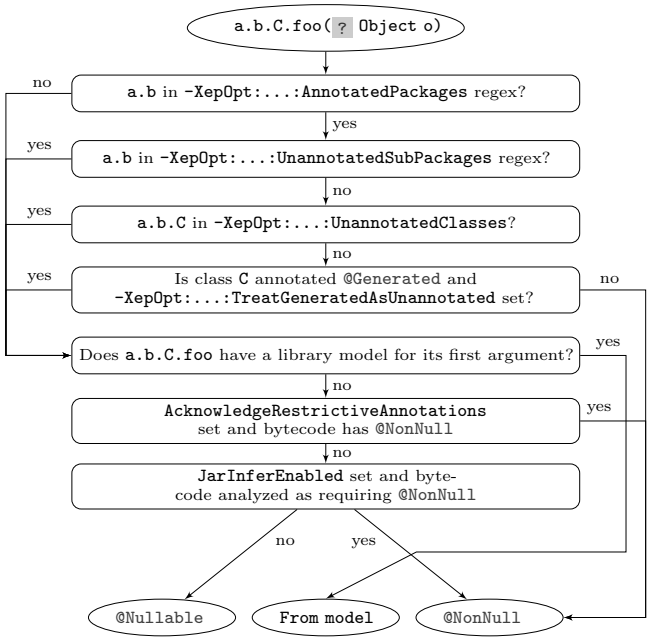

Figure 3: Flowchart for the treatment of unannotated code in NullAway.

Otherwise, the code is unannotated. NullAway then checks if there is a manually-written library model giving an annotation for the method argument; if so, that annotation is used. NullAway ships with 95 such library models, one per method and argument position pair. These mostly cover common methods from the JDK and Android SDK. NullAway can also load additional custom library models, but none of the open-source apps in our evaluation required it.

If the `AcknowledgeRestrictiveAnnotations` option is set, NullAway looks for explicit annotations within unannotated code and uses them if they are more restrictive than its default assumptions. This allows NullAway to opportunistically take advantage of explicitly-annotated third-party code, without forcing its default assumptions for checked code onto unannotated methods. Here, if `foo`'s parameter had an explicit `@NonNull` annotation, it would be used.

Finally, NullAway can leverage models automatically generated by JarInfer, a separate analysis we built for doing basic type inference on bytecodes. For example, if a method unconditionally dereferences its argument, JarInfer infers that the argument should be `@NonNull`. While JarInfer still only performs intra-procedural analysis on library entrypoints, we have found it useful at Uber for catching additional issues in interactions with libraries. A full description of JarInfer is outside the scope of this paper; we plan to extend it with greater functionality and present it in future work. None of the open-source apps we analyzed for our evaluation use JarInfer.

6 OTHER FEATURES

In this section we detail NullAway's handling of other Java features relevant to nullability, and describe some additional NullAway features that reduce false warnings.

Polymorphism NullAway does not yet support polymorphic nullability for generic types. Consider the following `Pair` type:

```
56  class Pair<T,U> {
57    public T first; public U second;
58  }
```

CFNullness allows for different uses of `Pair` to vary in nullability via type-use annotations on generic type parameters [28]. E.g., one can write `Pair<@Nullable String, String>` for `Pairs` where the first element may be null. In contrast, NullAway treats generic types like any other; for `Pair`, it assumes both fields have the `@NonNull` default. To allow `null` as a value, the fields themselves would need to be annotated `@Nullable`. Type-use annotations on the generic type parameters are ignored. This treatment is sound but could lead to undesirable code duplication; e.g., one may need to write a nearly-identical `FirstNullablePair`, `SecondNullablePair`, etc.

We have found this lack of support for type polymorphism to be only a minor annoyance thus far. A big mitigating factor is that most generic type usages in the Uber codebase are of `Collection` data structures, and it is a best practice to simply avoid storing `null` in such types [15]. However, we do see a need to eventually support type polymorphism, for cases like the above `Pair` type and also functional interface types like those in the `java.util.function` package. We plan to add support in future work, but doing so without compromising on build time overhead may require care.

Arrays NullAway unsoundly assumes that arrays do not contain `null` values. In contrast, CFNullness uses type-use annotations to reason about nullability of array contents; e.g., the type of an array of possibly-null `Strings` is written `@Nullable String []`. (Note that CFNullness does not soundly check array initialization by default [1, §3.3.4].) In the Uber code base, arrays of references are rarely used; `Collections` are used instead. For more array-intensive code, this NullAway unsoundness could have a greater impact.

Lambdas In Java, arguments to lambda expressions have neither explicit types nor annotations. NullAway uses the annotations of the single method in the functional interface corresponding to a lambda to determine the nullability of the arguments and return of the lambda. This is analogous to how the standard Java types of a lambda expression are inferred.

Handlers NullAway provides an internal extension mechanism called handlers. A handler implements a known interface to interpose itself at specific plug-in points during the analysis process and alter the nullability information available to NullAway. The following two features are implemented as handlers.

Pre- and post-conditions NullAway supports simple pre- and post-condition specifications via partial support

for `@Contract` annotations [7]. Here are some example usages of `@Contract` supported by NullAway:

```
public class NullnessHelper {
  @Contract("null -> false")
  static boolean isNonNull(@Nullable Object o) {
    return o != null;
  }
  @Contract("null -> fail")
  static void assertNonNull(@Nullable Object o) {
    if (o == null) throw new Error();
  }
  @Contract("!null -> !null")
  static @Nullable Object id(@Nullable Object o) {
    return o;
  }
}
```

The `@Contract` annotations document that `isNonNull` returns `false` when passed `null`, and that `assertNonNull` fails when passed `null`. The annotation on `id` indicates that if passed a non-null value, a non-null value is returned, yielding some support for parametric polymorphism (like `@PolyNull` in CFNullness [28]). Currently, NullAway trusts `@Contract` annotations, but we plan to add checking for them soon. We also plan to eventually add support for more sophisticated pre- and post-condition annotations like `@RequiresNonNull` and `@EnsuresNonNull` from CFNullness [1, §3.2.2].

Streams While NullAway's general type inference/refinement is strictly intra-procedural, handlers can propagate nullability information inter-procedurally for a few well-understood APIs. At Uber we use this functionality mostly for stream APIs such as RxJava [9]. Consider the following code using a common filter and map pattern:

```
public class Baz {
  @Nullable Object f;
  ...
}
public class StreamExample {
  ...
  public void foo(Observable<Baz> o) {
    o.filter(v -> v.f != null)
     .map(v -> bar(v.f.toString());
  }
}
```

In the above example, there are three separate procedures: `foo`, and the two lambdas passed to the `filter` and `map` method calls. The lambda for `filter` will filter out any value in the stream for which `v.f` is `null`. For the lambda inside `map`, NullAway's usual analysis would emit an error, due to the call of `toString()` on the `@Nullable` field `v.f`. But this code is safe, as objects with a null `f` field were already filtered out.

NullAway includes a handler that analyzes the exit statements of every lambda passed to `Observable.filter`, and the entry statement of every lambda passed to `Observable.map`. If the `map` call is chained immediately after the `filter` call (as in the previous example), this handler propagates the nullability information for the parameter of the `filter` lambda on exit (conditioned on the return value being `true`) to the parameter of the `map` lambda. In the example above, when the `filter` lambda returns `true`, `v.f` must be `@NonNull`. This fact gets preserved at the entry of the `map` lambda, and hence NullAway no longer reports a warning at the `toString()` call. This heuristic handles common cases observed in the Uber code base and reduces the need for warning suppressions, without introducing any new unsoundness.

7 IMPLEMENTATION AND DEPLOYMENT

NullAway is built as a plugin to the Error Prone framework for compile-time bug finding [2, 16]. Error Prone is carefully designed to ensure its checks run with low overhead, enabling the checking to run on every build of a project. Checks leverage the parsing and type checking already done by the Java compiler, thereby avoiding redundant work. Further, Error Prone interleaves running all checks in a *single pass* over the AST of each source file, a more efficient architecture than doing a separate AST traversal per check.

The core of NullAway primarily adheres to the single-pass architecture encouraged by Error Prone. Some additional AST traversal is required to collect class-wide information up front like which fields are `@NonNull`, to facilitate initialization checking (Section 3). To perform flow-sensitive type inference, NullAway uses the standalone Checker Framework dataflow library [24] to construct control-flow graphs (CFGs) and run dataflow analysis. CFG construction and dataflow analysis are by far the most costly operations performed by NullAway. The tool employs caching to ensure that dataflow analysis is run at most once per method and reused across standard type checking and initialization checking.

We profile NullAway regularly to ensure that performance has not regressed and have added several optimizations over time to remove inefficiencies. Regular performance measurement has been critical to keeping NullAway's overhead low, and we plan to add performance regression tests to help automatically catch changes that cause slowdowns.

NullAway has been deployed at Uber for nearly two years. For Android code, NullAway runs on every compilation (both local builds and during continuous integration), blocking any change that triggers a nullness warning from landing (unless the developer explicitly adds a `@SuppressWarnings` annotation). The tool runs on all Java source code in our monorepo, but not test code or third-party binary libraries. The latter are treated as unannotated code, with both restrictive annotations and JarInfer enabled (see Section 5).

8 EVALUATION

We evaluated NullAway on a diverse suite of open-source Java programs and via a long term deployment on the Android codebase for Uber. The evaluation targeted the following research questions:

RQ1: What is the annotation burden for using NullAway?

RQ2: How much build-time overhead does NullAway introduce, and how does this overhead compare to previous tools?

RQ3: Do NullAway's unsound assumptions for checked code lead to missed NPE bugs?

8.1 Experimental Setup

To evaluate NullAway's performance and effectiveness on open-source software, we gathered a diverse suite of benchmark Java projects from GitHub that already utilize NullAway. We searched for all projects integrating NullAway via the Gradle build system, and then included all non-duplicate projects that we could successfully build, excluding small demo projects. The projects vary in size and domain—they include widely-used Java, Android, and RxJava libraries, as well as some Android and Spring applications.

Table 1 summarizes the details of our benchmark suite, including the internal code base at Uber. Regarding RQ1, NullAway's annotation burden is quite reasonable, with 11.82 annotations per KLoc on the Uber code base, and 11.52 annotations per KLoc on the open-source benchmarks. As observed in previous work [28], using `@NonNull` as the default annotation both decreases the annotation burden and encourages better coding style.

Our experimental harness ran as follows. First, we ensured that all projects built without any warnings using the latest stable version of NullAway; the numbers in Table 1 include additional annotations required for a few cases. The harness captured the compiler arguments for each build target in each project based on Gradle's verbose output. Then it modified the arguments as needed to run each build with NullAway, CFNullness [12], and Eradicate [6]. We ran all tools in their default configuration; for NullAway the only preserved setting was the set of annotated packages.

To answer RQ2, we measured the overhead of each run against the time to run the standard Java compiler with no nullness checking. All experiments on the open-source apps were run on a single core of an Intel Xeon E5-2620 processor with 16GB RAM running Linux 4.4, and Java JDK 8. We used CFNullness v.2.8.1 and Infer v.0.15.0. Due to the size and complexity of Uber's build environment, we do not attempt to run other tools there; we still measure NullAway's overhead compared to compilation with it disabled.

To answer RQ3, we ran two different experiments. First, we studied all NPEs present in production crash data on Uber's applications over a period of 30 days, looking for cases where NullAway's unsound assumptions on checked code led to crashes. Uber's crash reporting infrastructure de-duplicates crash instances with the same stack trace. For the 30-day period we studied, there were 100 distinct stack traces involving NPEs. This includes crashes in both internal and production versions of the app, for all versions in use during the time period, possibly including months-old versions. We included all crashes to get the broadest dataset of NPEs in code that had passed NullAway's checks. Second, for the open-source benchmarks, we manually categorized a random subset of the additional warnings given by CFNullness, to see if they corresponded to real potential bugs missed by NullAway.

Data Availability NullAway and the scripts required to run our evaluation on the open-source benchmarks are already public. However, we cannot yet provide links to these resources, as they are difficult to anonymize. We will make these links public upon acceptance. We have provided our raw experimental data as supplementary material [10]. We cannot provide further data on Uber source code or crash reporting, as it is proprietary.

8.2 Compile Time Overheads

Figure 4 shows the build-time overheads of the tested nullness-checking tools as compared to compilation without nullness checking. On average, NullAway's build times are only 1.15× those of standard builds, compared to 2.7× for Eradicate and 5.9× for CFNullness. In fact, NullAway's highest observed overhead (1.39× for `uLeak`) is close to Eradicate's lowest (1.35× for `filesystem-generator`). Our supplementary materials [10] give full data on absolute compilation times and overheads for all runs.

We note that developers often perform incremental builds, in which only modified source files or targets and their dependencies are recompiled. Such builds are typically much faster than doing a clean rebuild of an entire project, so the overhead of nullness checking tools will consume less absolute time. Nevertheless, it is our experience that even with incremental builds, the overhead levels of Eradicate and CFNullness would still be a significant negative impact on developer productivity if run on every build.

For the NullAway deployment at Uber, measuring overhead is difficult due to use of modular and parallel builds, network-based caching, compilation daemons, and other annotation processors and analyses. As an estimate of overhead, we ran five builds of the entire monorepo with all forms of caching disabled, comparing our standard build with a build where NullAway is disabled, and we observed a 1.22× overhead on average.

To summarize: NullAway has *significantly lower compile time overhead* than previous tools, and hence can be enabled *for every build* on large codebases. By running on local builds with low overhead, NullAway helps developers catch potential NPEs early, improving code quality and productivity.

8.3 NullAway and NPEs at Uber

To aid in answering RQ3, Table 2 gives a categorization of NPEs observed in Uber Android apps over a 30-day period. NPEs were de-duplicated based on the full stack-trace of the exception, meaning that the same root cause can be counted multiple times (e.g., for a bug in a shared library used by multiple apps).

Ideally, we would like to compare the rate of NPEs at Uber before and after the introduction of static nullness checking. However, between a previous deployment of Eradicate and

Table 1: Benchmark Java Projects

Benchmark Name		KLoC	Annotations per KLoC		Description		
			Nullability	Suppression			
Uber repository		~3.3 MLoc	11.82	0.15	Android apps		
Open Source Projects	okbuck	8.96	13.06	0.67	Gradle plugin for Buck build system		
	caffeine	51.72	5.84	11.06	Libraries		Caching
	jib	27.14	15.59	0.04			Container image builder
	skaffold-tools	1.29	5.43	0.00			Containerization tools
	butterknife	15.55	3.47	0.06		Android	View injection
	picasso	9.56	11.61	0.21			Image caching
	RIBs	9.43	32.45	0.64			App architecture framework
	FloatingSpeedDial	2.21	28.51	0.00			UI library
	uLeak	1.38	3.62	2.90			Memory leak detection
	AutoDispose	8.27	5.32	0.48		RxJava	Stream handling
	ReactiveNetwork	2.16	0.00	6.02			Network monitoring
	keyvaluestore	1.40	11.43	0.00			Key-value store
	filesystem-generator	0.14	0.00	0.00			Filesystem manipulation
	QRContact	9.99	11.31	0.20	Apps	Android	Contact management
	test-ribs	6.29	24.64	0.95			App tutorial
	ColdSnap	5.13	24.37	0.00			Plant care app
	OANDAFX	0.99	46.46	0.00			Financial app
	meal-planner	2.62	1.15	0.00			Spring app: Planner

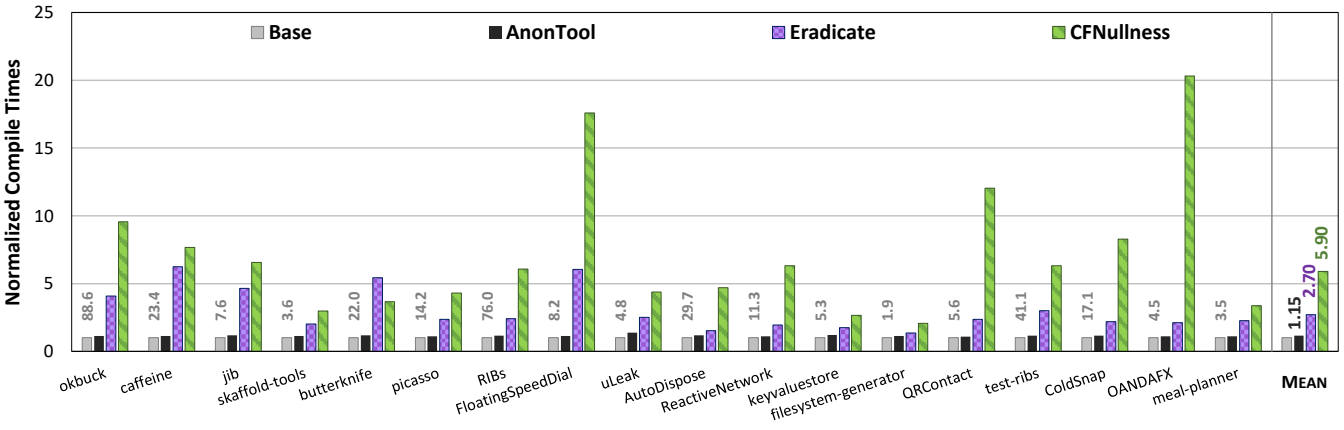

Figure 4: Build-time overheads of NullAway, CFNullness, and Eradicate. Compile times are normalized to the 'Base' compile times without nullness checking. Absolute times for Base compiles are labeled above the bars.

Table 2: Classification of NPEs from the Uber deployment

Category	Sub-category	Count
Unannotated library code	Android SDK / JDK	38
	Other Third-Party	16
	First Party Libs	10
	Total	**64**
Manual Suppressions	Precondition and assertion-like methods	14
	`@SuppressWarnings` annotations	3
	Total	**17**
Post-checking issues	Reflection	10
	Instrumentation	3
	Annotation Processor Misconfiguration	2
	Code stripping	2
	Total	**17**
Other		**2**
Total		**100**

NullAway itself, Uber's code base has been running some null checker for over 2 years, and we do not have data to do this comparison. We do note that the documented motivation for adopting these tools was the prevalence of NPEs as a category of crashes. Today, NPEs comprise less than 5% of observed crashes for our Android apps.

The most common type of NPEs missed by our tool (64%) are those involving unannotated third-party code. This case includes crashes within unannotated code and cases where unannotated methods returned `null` (NullAway optimistically assumes a non-null return value). Note that these cases were not necessarily bugs in the libraries; they could be a misunderstood API contract due to lack of nullness annotations. 38% of the crash stacks involved Android framework libraries or (rarely) the JDK classes, 16% involved other third-party libraries, and 10% involved Uber libraries outside the Android monorepo that do not build with NullAway.

The next broad category (17%) were manual suppressions of NullAway warnings. Only 3% were explicit `@SuppressWarnings` annotations, while 14% were calls to methods that perform a runtime check and crash when passed `null` (e.g. Guava's `Preconditions.checkNotNull` [5]). Such calls are warning suppressions since NullAway treats the argument as `@NonNull` after the call without proving that the call cannot fail.

An equally large category of crashes (17%), involved cases in which code was transformed in some way after checking. The most prevalent case was due to an object serialization framework unexpectedly writing `null` into a `@NonNull` field via reflection (10%). We also observed issues due to dynamic instrumentation breaking the assumptions made by the compiler (3%), misconfiguration of annotation processors resulting on incorrect or incomplete generated code (2%), and erroneous code stripping during the final steps of the build process (2%).

Finally, two uncategorized cases remain: one was an explicitly-thrown `NullPointerException` unrelated to nullability, and one could not be triaged correctly based on available data.

Critically, not a single NPE in this dataset was due to any of NullAway's unsound assumptions for checked code; they only related to handling of unannotated code or code transformations (e.g. reflection) that are out of the scope of nearly all static analysis tools. In other words, for this dataset, NullAway achieved its goal of *no false negatives in practice for checked code.*

8.4 Additional CFNullness Warnings

In our second experiment to answer RQ3, we inspected additional warnings emitted by CFNullness for build targets in the open-source benchmarks that passed NullAway with no warnings. As shown in Table 3, CFNullness raised a total of 404 additional warnings for 14 out of our 18 benchmark projects. We manually inspected 10 randomly-sampled warnings per benchmark (or all if there were fewer than 10), to determine if they were true bugs missed by NullAway (i.e., the code could cause an NPE at runtime).

Among the 122 randomly sampled warnings in Table 3, we found **no true NPE issues** that were missed by NullAway. Going left to right in Table 3, 30 warnings (25%) were due to stricter initialization checking in CFNullness, specifically prohibiting helper method calls from constructors and lack of support for the `@Initializer` annotation. 79 warnings (65%) were in calls to code deemed unannotated by NullAway. CFNullness treats third-party methods without annotations as having `@NonNull` parameters, whereas NullAway's optimistic treatment assumes `@Nullable` (see Section 5). We inspected these cases and did not find any true bugs, but as shown in Section 8.3 such bugs are possible. Also, some warnings were due to differences in built-in models of library code. For the "Other" column, one case involved sound assumptions around arrays (Section 6), and 7 others around purity and side-effects (Section 4). Finally, 5 of the issues (4%) would have generated NullAway warnings but were explicitly suppressed (last column).

To conclude, we found that on the open-source benchmarks, NullAway *did not miss any real NPE issues*, despite being unsound. Adding enough annotations to remove all CFNullness warnings would require significant effort, so in these cases, use of NullAway improved productivity without negative impact on safety.

8.5 Threats to Validity

The primary threat to the validity of our evaluation is our choice of benchmarks. We chose open-source benchmarks that already used NullAway, as getting unannotated code bases to compile with NullAway would have required significant effort. Also, NullAway was built for and tuned to the style of the Uber code base. Together these programs comprise millions of lines of code written by hundreds of developers, so we expect it to be representative of a large class of Java programs. But it is possible that code bases that have adopted NullAway are particularly suited to its checking style, and that on other types of programs (e.g., for scientific computing) NullAway's unsoundness would lead to more real-world NPEs.

Regarding build-time overhead, a threat is that our experimental configuration may not reflect typical build configurations, which e.g. may use concurrency or a daemon process (see discussion in Section 8.2). Our experience with the Uber deployment gives confidence that our measurements are reflective of typical builds.

9 RELATED WORK

Other sections of the paper have compared NullAway with CFNullness [21, 28] and Eradicate [6]. At Uber, Eradicate was used before NullAway was built, and it succeeded in greatly reducing the number of NPEs observed in production. The primary motivation for building NullAway was to reduce overhead such that checking could be run on every build (Eradicate only ran during continuous integration). Additionally, at the time NullAway was initially built, Eradicate was not able to process nullability annotations from bytecode for which it had not also processed the corresponding source code, leading to many missed warnings; this Eradicate issue has since been fixed.

NullAway's implementation was inspired by various partial nullability checks built into Error Prone [3, 4]. These checks also leverage dataflow analysis for inference via the Checker Framework dataflow library [24], from which NullAway's implementation borrows and extends. While Error Prone's checks can catch a variety of issues, they are not as complete as NullAway; e.g., they lack initialization checking (Section 3) and method override checking.

As mentioned in Section 3, checking of proper initialization has been the subject of a variety of previous work [18, 22, 23, 29, 30], as incorrect initialization jeopardizes reasoning about a many other object properties (e.g., immutability [31]). NullAway is distinguished by initialization checking that is

Table 3: Additional nullness warnings with CFNullness

Benchmark Name	# Warnings	Initialization Checking		Unannotated Code		Other	Suppressed by programmer annotations
		Helper methods	`@Initializer` methods	Optimistic defaults	Library models		
`okbuck`	17		1	6	3		
`caffeine`	47		3		5	2	
`jib`	141	2	1	4	1	2	
`skaffold-tools`	12		1		6	3	
`butterknife`	58	2	1	1	6		
`picasso`	31		2	4	4		
`RIBs`	22		4	3	2	1	
`FloatingSpeedDial`	5	1	2	2			
`AutoDispose`	21	3	1	5			1
`ReactiveNetwork`	6		1	2			3
`keyvaluestore`	7		1	6			
`QRContact`	4			4			
`test-ribs`	22	1	2	4	2		1
`meal-planner`	11	1		5	4		

unsound, yet has prevented nearly all initialization errors in a multi-million line code base over many months, requiring fewer annotations than the sound approaches.

Many approaches have been proposed for preventing NPEs with static analysis that does not require nullability annotations [19, 25–27]. Such approaches have the advantage of working out of the box on a subject code base, without requiring developers to initially add annotations. (In future work we hope to devise better type inference techniques to reduce this initial burden.) However, such approaches also have numerous drawbacks. Type checking for NullAway (and other major nullability type systems) is modular and efficient, leveraging type annotations at method boundaries. In contrast, most static analysis approaches are not modular, forcing an expensive global re-analysis at each code change. The bi-abduction approach of Facebook Infer is modular [19] but still significantly more complex and expensive than type checking.

Beyond analysis running time, null pointer issues discovered via static analysis can be difficult for a developer to understand, as they can involve tracing inter-procedural data flow. In contrast, NullAway errors always relate to a local violation of the typing discipline, and hence can be understood by only looking at surrounding code and type signatures of other methods. Further, nullability annotations serve as a form of compiler-checked documentation and can make code easier to read and maintain. Overall, we believe that if developers can put in the initial effort to adopt type-based nullability checking, it provides an overall better developer experience than relying on pure static analysis.

Section 8 showed that though NullAway is unsound, it has been able to prevent nearly all NPEs in production Android code at Uber. The RacerD system for static data race detection [17] also aims for this threshold of few to no false negatives in practice, based on bugs observed in the field. We anticipate that future work will continue this trend of static analysis and type system design based on preventing nearly all errors previously observed in testing and production, rather than aiming for strict soundness.

10 CONCLUSIONS

We have presented NullAway, a practical tool for type-based null safety of large-scale Java applications. NullAway has much lower build-time overhead than previous tools, enabling null checking on every build. Further, NullAway's various checks have been carefully tuned to minimize false negatives in practice while imposing a reasonable annotation burden. Real-world crash data from Uber showed that NullAway's unsound assumptions did not lead to *any* NPEs within checked code. NullAway runs on every build of millions of lines of code at Uber and has been adopted by many other open-source projects and companies.

REFERENCES

[1] 2019. Checker Framework Manual. https://checkerframework.org/manual/. Accessed: 2019-01-29.
[2] 2019. Error Prone. http://errorprone.info/ Accessed: 2019-02-07.
[3] 2019. Error Prone NullableDereference check. https://git.io/fhQkO. Accessed: 2019-01-29.
[4] 2019. Error Prone ReturnMissingNullable check. https://git.io/fhQk3. Accessed: 2019-01-29.
[5] 2019. Google Core Libraries for Java (Guava). https://github.com/google/guava Accessed: 2019-02-10.
[6] 2019. Infer : Eradicate. https://fbinfer.com/docs/eradicate.html Accessed: 2019-01-29.
[7] 2019. IntelliJ IDEA @Contract. https://www.jetbrains.com/help/idea/contract-annotations.html Accessed: 2019-02-07.
[8] 2019. Kotlin Programming Language. https://kotlinlang.org/. Accessed: 2019-01-29.
[9] 2019. ReactiveX/RxJava. https://github.com/ReactiveX/RxJava Accessed: 2019-02-10.
[10] 2019. Supplementary Data. https://figshare.com/s/a212932795a43c377a3f. Accessed: 2019-02-20.
[11] 2019. Swift Programming Language. https://swift.org/. Accessed: 2019-01-29.
[12] 2019. The Checker Framework. https://github.com/typetools/checker-framework Accessed: 2019-01-29.
[13] 2019. The Java Language Specification. https://docs.oracle.com/javase/specs/jls/se9/html/. Accessed: 2019-01-29.
[14] 2019. Understand the Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed: 2019-01-29.
[15] 2019. Using and Avoiding Null Explained. https://github.com/google/guava/wiki/UsingAndAvoidingNullExplained. Accessed: 2019-01-29.
[16] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 14–23. https://doi.org/10.1109/SCAM.2012.28

[17] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *PACMPL* 2, OOPSLA (2018), 144:1–144:28. https://doi.org/10.1145/3276514

[18] Dan Brotherston, Werner Dietl, and Ondrej Lhoták. 2017. Granullar: gradual nullable types for Java. In *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*. 87–97. https://doi.org/10.1145/3033019.3033032

[19] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[20] Alain Deutsch. 1994. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 230–241. https://doi.org/10.1145/178243.178263

[21] 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

[22] Manuel Fähndrich and K. Rustan M. Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*. 302–312. https://doi.org/10.1145/949305.949332

[23] Manuel Fähndrich and Songtao Xia. 2007. Establishing object invariants with delayed types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 337–350. https://doi.org/10.1145/1297027.1297052

[24] Stefan Heule and Charlie Garrett. 2019. A Dataflow Framework for Java. https://checkerframework.org/manual/checker-framework-dataflow-manual.pdf Accessed: 2019-02-07.

[25] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Gowri Nanda. 2008. Verifying dereference safety via expanding-scope analysis. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. 213–224. https://doi.org/10.1145/1390630.1390657

[26] 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

[27] Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate Interprocedural Null-Dereference Analysis for Java. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 133–143. https://doi.org/10.1109/ICSE.2009.5070515

[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] Xin Qi and Andrew C. Myers. 2009. Masked types for sound object initialization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 53–65. https://doi.org/10.1145/1480881.1480890

[30] Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: a lightweight type system for object initialisation. 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*. 1013–1032. https://doi.org/10.1145/2048066.2048142

[31] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and immutability in generic Java. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 598–617. https://doi.org/10.1145/1869459.1869509