Safe Stream-Based Programming with Refinement Types

Benno Stein
University of Colorado Boulder
Boulder, Colorado, USA
benno.stein@colorado.edu

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

Manu Sridharan
Uber Technologies, Inc.
San Francisco, California, USA
msridhar@uber.com

Bor-Yuh Evan Chang
University of Colorado Boulder
Boulder, Colorado, USA
evan.chang@colorado.edu

ABSTRACT

In stream-based programming, data sources are abstracted as a stream of values that can be manipulated via callback functions. Stream-based programming is exploding in popularity, as it provides a powerful and expressive paradigm for handling asynchronous data sources in interactive software. However, high-level stream abstractions can also make it difficult for developers to reason about control- and data-flow relationships in their programs. This is particularly impactful when asynchronous stream-based code interacts with thread-limited features such as UI frameworks that restrict UI access to a single thread, since the threading behavior of streaming constructs is often non-intuitive and insufficiently documented.

In this paper, we present a type-based approach that can statically prove the thread-safety of UI accesses in stream-based software. Our key insight is that the fluent APIs of stream-processing frameworks enable the tracking of threads via type-refinement, making it possible to reason automatically about what thread a piece of code runs on – a difficult problem in general.

We implement the system as an annotation-based Java typechecker for Android programs built upon the popular ReactiveX framework and evaluate its efficacy by annotating and analyzing 8 open-source apps, where we find 33 instances of unsafe UI access while incurring an annotation burden of only one annotation per 186 source lines of code. We also report on our experience applying the typechecker to two much larger apps from the Uber Technologies, Inc. codebase, where it currently runs on every code change and blocks changes that introduce potential threading bugs.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; **Formal software verification**;

KEYWORDS

stream-based programming, refinement types, mobile applications

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238174

ACM Reference Format:
Benno Stein, Lazaro Clapp, Manu Sridharan, and Bor-Yuh Evan Chang. 2018. Safe Stream-Based Programming with Refinement Types. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238174

1 INTRODUCTION

Many popular user interface frameworks (e.g., Swing, Cocoa, Eclipse, iOS, Android) distinguish a single main thread from which all UI accesses must be performed [4, 5, 16, 25, 47]. This design is preferred by library developers since it eliminates the need for library-internal synchronization: there is no need to worry about data races or deadlock when only one thread is allowed to perform UI operations.

However, the single UI thread model requires application developers to carefully avoid interacting with the UI from other threads since doing so results in a runtime crash or undefined behavior.

Such *invalid thread access* bugs are very common in practice: a Google search for Android's `CalledFromWrongThreadException`, one of several exceptions that Android can throw when the UI is accessed improperly, yields over 47,000 results, including numerous Github bug reports, StackOverflow questions, and developer guides and tutorials.

Furthermore, invalid thread accesses are difficult to detect and debug in practice. Existing UI testing techniques are often unable to achieve adequate coverage of possible UI interaction traces [11] and struggle with bugs that only manifest on certain devices in the diverse Android hardware and software ecosystem [19, 58]. Program analysis approaches to finding improperly threaded UI accesses are similarly inadequate: the callgraph-reachability technique proposed by Zhang et al. [64] and the effect type system of Gordon et al. [28] both identify methods that interact with the UI effectively but use a very conservative and restrictive model to determine when those methods run on the UI thread; on the other hand, general approaches to concurrency analysis typically focus on shared memory access rather than determining the thread on which a given piece of code will run [20, 44].

In recent years, there has been an explosion in popularity of stream-based programming frameworks like Reactive Extensions [50], especially for interactive software like Android applications that need to respond to user input in real time. Such frameworks provide expressive and convenient threading abstractions for manipulating streams of data and computation, but offer little in the

way of tool support to help developers avoid invalid UI access by reasoning about what thread a stream is running on.

We propose in this paper a refinement type-based static analysis that identifies possible invalid thread UI accesses in stream-based Android applications, combining an effect type system that tracks the possible UI interactions performed by methods with a thread type system that encodes the possible threading behaviors of asynchronous data streams.

Our approach not only detects (or proves the absence of) improper UI access, but also helps developers understand and reason about UI interactions in their code. Furthermore, our annotation-based approach makes that reasoning explicit and self-documenting, allowing future contributors to more easily understand, modify, or extend previously annotated code.

Our typechecker currently runs on every commit made to two major Android applications at Uber Technologies, Inc. (Uber), blocking any changes that may introduce invalid thread access bugs. In practice, we have found that the typechecker catches potential bugs earlier in the development process and more efficiently than existing testing and manual code review are able to.

We build on the insights of Gordon et al. [28], who first showed the applicability of effect-typing to modern UI frameworks with a distinguished UI thread, by extending their work in two key dimensions. First, we introduce a thread type system for determining the thread on which stream operators execute. Combining the thread type system with the effect typing of Gordon et al. enables verifying UI effect safety for a wider class of threading constructs, with almost no additional annotation burden for developers. Second, we develop an effect inference technique for callbacks and lambda abstractions that further lowers the annotation burden of UI effect typing and improves code readability.

Contributions: The primary contributions of our work are as follows:

- We introduce a novel refinement-type system that soundly verifies that stream-processing code only accesses the UI from streams running on the Android main thread. We demonstrate its efficacy by implementing a static annotation-based type checker for Android applications built upon the ReactiveX Java stream-based programming framework. Our system statically refines the types of callbacks and lambda abstractions by the *effects* they may incur and data/event streams by the *threads* on which they may run, then verifies that non-UI thread streams never call methods with UI effect.
- We analyze a corpus of 8 open-source Android applications, as well as two large closed-source applications developed at Uber. In doing so, we find that *(a)* improper UI access is detectable by our tool and prevalent in both open-source and closed-source codebases and *(b)* annotation burden on the programmer is low enough for the tool to be incorporated into a production developer workflow. In total, we find 33 instances of UI-effectful callbacks running on non-UI-threaded streams in the open-source corpus. At Uber our tool runs on every code change as part of continuous integration, blocking any change that may introduce stream-based threading bugs.

2 OVERVIEW

In this section, we provide background information about stream-based programming frameworks, the Android UI model, and refinement typechecking by applying our tool to a simple example from a user's perspective.

2.1 Reactive Extensions

Reactive Extensions (ReactiveX) [50] is a multi-language framework for asynchronous stream-based programming which allows developers to easily write code that operates over streams of events or data, composing and transforming them with various functional operators and subscribing callbacks to perform computations in response to events.

Stream processing frameworks have gained popularity in recent years due to their ability to provide a uniform interface to multiple asynchronous input sources, allowing developers to build responsive interactive applications and easily interoperate (via the stream API) with new frameworks and technologies. Stream-based programming also encourages a functional programming paradigm, preferring composable modular computations to imperative procedures over mutable state.

A typical use-case of ReactiveX is to receive or generate an `Observable` stream, perform a number of operations that modify its data or thread, and then subscribe an `Observer` (or other callback-like object) to asynchronously consume the resulting event stream. This so-called "fluent" interface, in which multiple calls are chained together, is a hallmark of the stream-based programming paradigm, combining ideas from the `Observer` pattern, the `Iterator` pattern, and functional programming [50].

Take, for example, the code snippet in Fig. 1, which updates car locations on a map using the ReactiveX framework. The `carLocationData` stream represents location data for some set of cars, updated periodically by a remote server. The `filter` operator filters the stream to include only cars currently without a passenger. Then, `observeOn` moves subsequent operations to the main thread, a requirement for performing UI updates on Android. The `delay` operator introduces a delay before each location update, to allow for other processing to complete. Finally, the anonymous function passed to `subscribe` invokes UI APIs to display the cars in the map.

A key feature of the ReactiveX fluent interface is that each operation in a chain of calls returns a new `Observable` instance rather than performing side effects on the receiver of the call. This side-effect free nature of ReactiveX enables a type-based analysis, since each intermediate `Observable` instance in the call chain can be given a single static type that is not subject to change later in the chain. Note that ReactiveX's API is distinct from the Builder pattern [23], which supports a similar call-chaining syntax but does perform side effects with each call, passing the same Builder instance through the chain.

Though Fig. 1's example makes use of only a few simple stream operators, ReactiveX provides a wide range, from functional programming standards like `filter` (emit only those data that satisfy a given predicate) to more exotic combinators like `switchMap` (map incoming data to new streams, emitting events only from the most recent datum's stream). Streams (i.e. `Observables`) possess 340 such

```
Observable<...> carLocationData = ... ;
carLocationData
 .filter(car -> /*car has no passenger*/)
 .observeOn(AndroidSchedulers.mainThread())
 .delay(100, TimeUnit.MILLISECONDS)
 .subscribe(
   car -> { /* display car on map */ },
   err -> { /* render error message */ })
```

Figure 1: Simple example usage of ReactiveX `Observable` streams. Contains the distillation of a threading bug that was detected by our tool and fixed, as explained in section 2.2.

operators in ReactiveX Java version 2.1.12, in addition to any custom operators defined by third party libraries or more specialized types of streams.

With the expressivity of such a large and complex framework inevitably comes a steep learning curve, since it is difficult for developers to become familiar with the API. This contributes to the preponderance of threading bugs in real-world applications using ReactiveX. In our experiments, we find that many programs contain latent threading bugs that require a nuanced understanding of the framework to detect.

The program in Fig. 1, for example, looks safe from improper UI access at first glance: its author was careful to `observeOn` the Android main thread before subscribing a callback that renders UI elements and error messages. However, due to the threading behavior of the `delay` operator, the program actually accesses the UI from a background thread.

2.2 Refinement Typechecking

In spite of the intractability of determining what thread a piece of code will run on in general, stream-based frameworks like Reactive Extensions are amenable to thread analysis by means of *refinement types* [22, 48]. As discussed in the previous section, the fluent functional interface of ReactiveX streams is the key feature which enables the use of a type-based approach to track the thread of intermediate calls in the chain, obviating the need for more expensive general-purpose thread analyses.

Informally, a refinement type system can be thought of as augmenting *base types* (e.g., integers, lists, strings) by *qualifiers* that further restrict the values of the base type (e.g., *positive* integers, *nonempty* lists, *ASCII* strings). Refinement subtyping holds when both base subtyping and qualifier subtyping hold, and standard intuitions about nominal subtyping over base types behave analogously for refined types[1].

In this paper, we verify the safety of UI access in stream-based Android programs using two separate but parallel type refinements.

First, we refine the types of all methods and callback-like objects with an *effect qualifier* from Fig. 2, which places an upper bound on the side effects they may incur, as in Gordon et al. [28]. We say a method is "UI-effectful" when it may access the UI and is annotated with `@UIEffect`.

[1]Namely, refinement subtyping is reflexive and transitive, and the Liskov substitution principle holds for refined types.

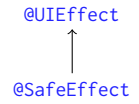

Figure 2: Qualifier hierarchy for effect type refinements.

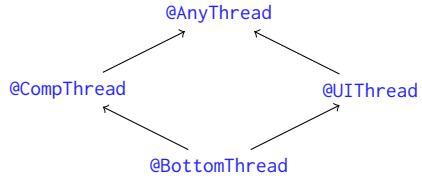

Figure 3: Qualifier hierarchy for thread type refinements.

Next, we refine the types of all data streams with a *thread qualifier* from Fig. 3, which places an upper bound on the thread on which they may emit events. If such a thread cannot be determined statically, the stream will have the trivial `@AnyThread` qualifier, which does not restrict the base type at all.

In both cases, we take care to minimize annotation burden using sensible default qualifiers, shorthands to annotate entire classes or packages, and type inference to determine qualifiers where they are implied.

Finally, we typecheck the program, confirming that all assignments, return values, and function arguments respect their declared types, UI-effectful methods are never called from safe-effectful methods, and UI-effectful callbacks are subscribed only to UI-threaded streams. Otherwise, we report warnings to the developer where these conditions are not met.

A more precise and formal treatment of the refinement type system we instantiate can be found in section 3. For now, we will build intuition by applying our refinement typechecker to the program in Fig. 1.

When a developer tries to compile the program in Fig. 1, our compiler issues the following error at the `subscribe` callsite:

```
error: [rx.thread.violation] Subscribing a callback with
@UIEffect to an observable scheduled on @CompThread; @UIEffect
effects are limited to @UIThread observables
```

This error indicates a potential threading error in the program: the callbacks being passed to `subscribe` can touch the UI, but they are being subscribed to a `Observable` scheduled on a background computation thread.

The reason for this error is that `delay` automatically returns a stream running on a background computation thread. Thus, even though the developer used `observeOn` to force the stream onto the main thread before subscribing UI-effectful callbacks, the intermediate call to `delay` promptly moved it to a computation thread. In practice, we found that even expert developers were often unaware of this `delay` behavior, documentation of which is limited to a short note buried deep in a large generated documentation file.

However, our typechecker refines the types of each program value: `@AnyThread` for the initial `carLocationData` stream, `@AnyThread` for the result of `filter`, `@UIThread` for the result of `observeOn`, `@CompThread` for the result of `delay`, and `@UIEffect` for the lambda expressions passed to `subscribe`. Thus, our typechecker is able

to identify the bug – subscription of a `@UIEffect` function onto a `@CompThread` stream – and issues the error reproduced above.

The fix to this issue, once the developer has been made aware of it by the typechecker, is simple and requires no annotations: swap the positions of the `delay` and `observeOn` calls and the code will typecheck and compile without error.

3 THREAD & EFFECT SEMANTICS

This section details the thread semantics of the Android UI framework and the ReactiveX threading model and formalizes the thread and effect type systems we use to analyze ReactiveX-based Android applications.

Recall that the Android UI toolkit is not thread-safe and adheres to a single thread model; as such, any code that accesses UI components *must* do so from the UI thread [25]. This is a relatively standard model for other UI frameworks, including iOS as well as Java's Swing, SWT, and AWT [4].

Existing work has established *effect types* as a useful abstraction for developers to avoid violating this single-thread assumption [28].

However, such work is limited to relatively simple threading models where the library provides an interface to the UI thread for code running elsewhere: for example, the Eclipse SWT UI framework defines a function with signature

```
static void asyncExec(Runnable r);
```

which allows a developer to pass some UI-effectful code `r` to be run on the UI thread.

While Android does provide analogous functions (e.g., `Activity#runOnUiThread`, `View#post`) that can be analyzed by existing effect-typing techniques, it also provides more expressive functions beyond their reach (e.g., `AsyncTask`, `Handler`). In addition, Reactive Extensions' threading constructs introduce even more complexity and require new techniques to be analyzed properly.

3.1 Effects

The application of effect type systems to UI frameworks is a fairly well-understood technique: functions are annotated with effect qualifiers, which can then be checked to verify that the annotated function does not perform any effects not permitted by its annotation. Our effect system builds on top of that of Gordon et al. [28], wherein functions have one of two effect annotations: `@UIEffect`, denoting a method that may (or may not) interact with the UI, or `@SafeEffect`, denoting a method that is guaranteed not to touch the UI[2].

The sub-effecting relation $\preceq$ is given by `@SafeEffect` $\preceq$ `@UIEffect` and the two reflexive relationships, `@SafeEffect` $\preceq$ `@SafeEffect` and `@UIEffect` $\preceq$ `@UIEffect`. That is, a method with safe effect can be used in place of a method with UI effect, but not vice versa.

We say that a program is *effect-safe* when its effect annotations over-approximate all possible effects performed at runtime; thus, in an effect-safe program, a method annotated `@SafeEffect` is guaranteed never to interact with the UI. Checking effect-safety of a program whose methods are annotated by their effect reduces to checking the following two conditions:

- *Transitivity:* A method with effect annotation e may only call a method with effect annotation e' if $e' \preceq e$.

[2]Note that this ignores polymorphic qualifiers, which will be detailed in section 3.3.

```
class A {
  @UIEffect   void foo() {...}
  @SafeEffect void bar() {...} }
class B extends A {
  // Transitivity violation
  @SafeEffect void baz() { foo(); }
  // Inheritance violation
  @UIEffect void bar() {...} }
```

Figure 4: Example violations of the effect type system. `B#baz` violates the transitivity condition because it is annotated as safe but calls a UI-effectful method `A#foo`, while `B#bar` violates the inheritance condition because it manipulates the UI but overrides a method declared to be safe.

- *Inheritance:* A method with effect annotation e may only override a method with effect annotation e' if $e \preceq e'$.

In other words, the *transitivity* condition states that safe methods cannot call UI methods, while the *inheritance* condition states that methods cannot have UI effect if they override a safe method. Assuming that UI-effectful library methods are annotated accordingly, proving these two conditions suffices to show that the UI is only accessed from methods annotated with `@UIEffect`. Figure 4 provides concrete examples of effect-safety violations.

The reader may find it useful to think of these two conditions in terms of reachability in a directed graph whose vertices are methods, with edges from callers to callees and from superclass methods to overriding subclass methods. In such a graph, an edge from a method with effect e to a method with effect e' violates one of the above conditions when $e \prec e'$. The task of applying effect types to an Android application is thus equivalent to determining the region of nodes from which Android UI methods are reachable: that region has UI effect, while its complement has safe effect.

Other than the lambda support and inference mechanism described in section 3.4, this effect type system is identical to that of Gordon et al. [28].

3.2 Threads

The threading behavior of Android applications that make use of stream-based programming frameworks like Reactive Extensions is determined not only by a small set of Android API methods with fixed semantics but also by a wide range of stream operators with dynamic threading behavior. Effect typing alone is therefore insufficient to properly verify the UI thread-safety of such applications.

Consider, for example, the `subscribe` method of `Observable`, which is called on a stream in order to register some callback to be executed whenever an event is emitted by the receiver stream.

In contrast to Android's `runOnUiThread`, which can safely be passed a UI-effectful callback in all contexts, `subscribe` can only be passed a UI-effectful callback `obs` when the receiver stream is running on the UI thread.

In order to express that invariant, an analysis must reason not only about the effects of methods, but also the threads on which streams emit events (and, by extension, execute subscribed callbacks).

To that end, we augment our type system with type annotations that refine stream types by their thread. These type annotations are drawn from the qualifier hierarchy given in Fig. 3. The top of the qualifier hierarchy, `@AnyThread`, denotes a stream that can emit events on any thread; `@UIThread` and `@CompThread` denote streams that can emit events only on the UI thread or a background computation thread, respectively; `@BottomThread` denotes a stream that cannot emit events on any thread. This bottom type is never written by a programmer but is used within the typechecker for the `null` value, dead code, and wildcard lower bounds [9].

Annotating stream operators with thread type refinements allows a typechecker to reason about the threading behavior of those constructs. For example, the thread semantics of the `delay` function used in the motivating example in Fig. 1 can be specified by annotating its receiver `@AnyThread` and its return type `@CompThread`.

Combining thread refinement types for streams with effect refinement types for methods is the key idea that allows our typechecker to verify UI thread-safety of stream-based Android applications by checking that subscription of UI-effectful callbacks only occurs on UI-threaded streams.

3.3 Qualifier Polymorphism

Some design patterns – particularly those designed for modularity and reusability – have effect and thread behavior that cannot be expressed by a single type signature with fixed refinements. In these cases, we make use of *qualifier polymorphism*.

Qualifier polymorphism is a form of parametric polymorphism, which also underlies generics in Java and C# and universally quantified types in Haskell and OCaml. Consider, for example, this method from the Java collections library, which creates a singleton set:

```
Set<T> singleton(T obj){...}
```

The generic type variable `T` may be instantiated as any single Java object type, constraining the element type of the returned set to be the same as the argument type.

Similarly, qualifier polymorphism uses a generic refinement variable to *relate* type refinements rather than explicitly annotating types with a fixed qualifier. We define a `@PolyThread` (resp., `@PolyUIEffect`) qualifier that may be instantiated as any concrete thread (resp., effect) qualifier, constraining the refinements on multiple types to be the same[3].

Qualifier polymorphism is well suited to several design patterns in stream-based Android applications, several examples of which are selected and reproduced in Figure 5.

The `Callback` interface exhibits effect polymorphism: we use the polymorphic qualifiers `@PolyUIType` and `@PolyUIEffect` to enforce that a `Callback` instance has a UI annotation when its `handleMessage` method has UI effect. Similar interfaces such as `Runnable`, `Action`, and `Observer` are annotated analogously, relating the refinement type of the callback-like object to the effect of its implemented method(s).

[3] Implicitly, a class or method with polymorphic qualifier annotations is parameterized by a single refinement type variable which is used wherever the polymorphic qualifier is written. As such, it is impossible to parameterize a definition by multiple refinement type variables, but we have not found any any code patterns in practice where such a type is required. This polymorphic qualifier syntax is defined and provided by the Checker Framework [9].

```
@PolyUIType interface Callback {
  @PolyUIEffect
  boolean handleMessage(Message m); }

class Observable<T> {
  @PolyThread Observable<T> take
   (@PolyThread Observable<T> this,
    int k){...};
  @PolyThread Observable<T> observeOn
   (@PolyThread Scheduler thread){...};}
```

Figure 5: Examples of thread- and effect-polymorphic types, drawn from Reactive Extensions' `io.reactivex.Observable` and Android's `android.os.Handler`, respectively.

Methods that take `Callback` instances with UI effect versions of `handleMessage` will need to declare the corresponding formal parameter as `@UI Callback`. Methods that do not care about the callback's effect take `@PolyUI Callback` instances. Our tool defaults to interpreting unannotated formals of a polymorphic type as `@AlwaysSafe` (non-UI affecting) instances. Analogous logic applies to the types of fields and locals.

The `Observable` class – the main stream data type in ReactiveX – exhibits two distinct forms of thread polymorphism. First, most of its methods (e.g., `take` in Fig. 5) do not affect the thread of the stream they operate on; we express this behavior by constraining their receiver[4] and return values to have the same thread refinement with the `@PolyThread` qualifier. Second, we use thread polymorphism to express the dependently-typed behavior of the `observeOn` operator, which takes a `Scheduler` (e.g., a thread pool) and returns a stream emitting events on that thread pool. In order to do so, we overload the meaning of our thread qualifiers to apply to schedulers as well as streams and annotate the scheduler and the returned stream of `observeOn` with the `@PolyThread` qualifier.

Qualifier polymorphism introduces some additional complexity to the definition of effect-safety given in section 3.1. The interaction between polymorphic and concrete effects is fairly straightforward: we have `@SafeEffect` $\preceq$ `@PolyUIEffect` and `@PolyUIEffect` $\preceq$ `@UIEffect`, since those relations would hold for whatever concrete effect `@PolyUIEffect` takes on. That is, the body of an effect-polymorphic method can call any method with safe effect but is forbidden from calling UI-effectful methods. On the other hand, the body of an effect-polymorphic method may only call other effect-polymorphic methods when they share the same receiver object, since arbitrary other effect-polymorphic methods may be instantiated with an incompatible concrete effect.

3.4 Lambdas and Qualifier Inference

The callbacks used in stream processing frameworks like ReactiveX are typically written as either anonymous inner classes or Java 8 lambda expressions. When a callback is passed to a method whose formal parameter is annotated with a particular effect qualifier, the anonymous class or lambda must itself have a compatible type qualifier. In the case of anonymous inner classes this can be achieved by

[4] The receiver `this` is explicitly written and annotated as the first argument to the method, as per Java's Type Annotation specification; this does not affect the semantics of the function whatsoever [36].

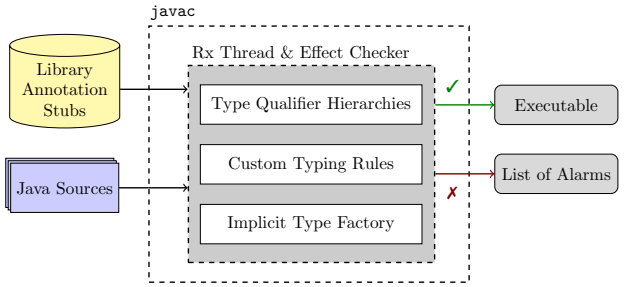

Figure 6: Typechecker infrastructure diagram showing its internal components, inputs and outputs. From a user's perspective, UI thread-safety checking is integrated seamlessly with other `javac` compile-time checks in this default configuration, but it is also possible (by means of a command-line option) to run the checker as a standalone process, emitting a success message instead of generating an executable when the program is deemed safe.

using a type-use annotation (e.g., `new @UI Consumer{...}` to specify a `Consumer` whose `accept` method has `@UIEffect`). However, the syntax of Java lambda expressions does not permit any explicit type annotations (refinement or otherwise). Rather, their type is resolved through *type inference*.

Consider, for example, the Java type of the lambda expression passed as the first argument to `Observable#subscribe` in Figure 1:

```
car -> { /*display car on map */}
```

The first formal parameter of `Observable#subscribe` has type `@PolyUI Consumer`, so the compiler infers that `Consumer` is the *functional interface*[5] base type of the lambda. During compilation, `javac` will convert any lambda expression to an anonymous instantiation of a compatible functional interface, inferring its base type from the context.

However, the base type inference mechanism does not apply to refinement type qualifiers. Instead, we apply local type qualifier inference to compute the proper effect annotation with which to instantiate the `@PolyUI` polymorphic qualifier by inspecting the body of the lambda expression. If a call to a method with `@UIEffect` effect is found within the body of the lambda, we mark the corresponding anonymous instance of the functional interface as `@UI`, and otherwise as safe. In the example above, as long as the code in brackets includes at least one call to an `@UIEffect` method, we infer the type of the lambda to be `@UI Consumer`.

4 TYPECHECKER IMPLEMENTATION

This section details the implementation of a typechecker for the aforementioned thread and effect type systems, which is able to soundly verify the UI thread safety of ReactiveX-based Java Android applications.

The typechecker's design balances competing goals: it must not only provide reliable guarantees of safety and correctness, but also be easy-to-use and integrate into developer workflows. What's more, in order to find and fix existing UI threading bugs, it must be applicable to legacy codebases without a prohibitive amount of configuration or annotation effort. To that end, we build the typechecker upon the Checker Framework [9, 14, 48], which provides infrastructure for building custom typecheckers that augment Java's base type system.

Refinement types are implemented as Checker Framework type qualifiers: annotations on existing Java base types which are then processed at compile-time by our typechecker. The Checker Framework allows users to check those types by simply invoking a `javac` compiler with certain command-line flags. This design enables any modern Java build system or IDE to incorporate UI thread-safety typechecking with minimal configuration overhead.

The typechecker plugin itself consists primarily of three components, as shown in Fig. 6: the type qualifiers themselves, custom typing rules that specify the UI thread-safety invariant, and a type factory that generates annotations that are not explicitly written but rather inherited, derived, or inferred.

The *type qualifier hierarchies* – described in detail individually in Section 3 – implement Java type annotations for each thread and effect type. In addition to the type hierarchies shown in Figures 2 and 3, this also includes polymorphic qualifiers that range over those lattices as well as class- and package-level annotations which apply some annotation to each method therein.

The *custom typing rules* express the constraints on effect inheritance and transitivity and enforce the core UI thread-safety invariant: that `@UIEffect`-ful callbacks may only be subscribed to `@UIThread` streams. These custom typing rules augment the standard rules implemented by the Checker Framework which constrain, for example, actual parameters to declared formal parameter types, assignment *r*-values to corresponding *l*-value types, and return values to declared method signatures.

Finally, the *implicit type factory* is responsible for generating type refinements for those variables that lack an explicit refinement annotation. This process takes one of three forms:

- *Inheritance:* In addition to the `@UIEffect` and `@SafeEffect` method annotations, the effect type system we build upon also provides shorthand annotations to facilitate blanket annotation of all methods in a class or package [28]. These shorthands are especially useful when all methods of a particular package or class share the same effect, saving the effort of annotating each one manually.
- *Inference:* Whenever our typechecker encounters a lambda expression, we retrieve the functional interface base type inferred by `javac` for the lambda. If that type has a concrete effect qualifier, no inference is required and we simply apply that qualifier. Otherwise, we scan the body of the lambda for any invocations of methods with `@UIEffect`. If any are found, then the anonymous instance associated with the lambda expression is annotated `@UI` (see Section 3.4).
- *Defaults:* When an un-annotated method or variable does not inherit a type refinement and inference does not apply, a sensible default is chosen by the typechecker. These defaults minimize annotation burden dramatically by only forcing developers to write annotations where the type refinement varies from the default.
  We refine un-annotated methods with the `@SafeEffect` qualifier so that annotations only need to be explicitly written on code that interacts with the UI, while un-annotated streams

[5] A functional interface is any Java interface with a single abstract method.

receive the `@AnyThread` trivial refinement so that, unless otherwise specified, we soundly assume a stream could emit events on any thread.
The choice of default annotations is a design choice; in practice, we find that these defaults largely correspond to developer intuitions and reduce annotation overhead.

Those three elements – qualifier hierarchies, custom typing rules, and implicit type generation – suffice to typecheck *whole* programs: programs that are complete and self-contained in a set of source files. The stream-based Android apps we are concerned with, however, are *open* programs which interact with other libraries and frameworks whose source code the typechecker may not have access to: ReactiveX and the Android standard library, at a bare minimum.

In order to typecheck code that interacts with un-analyzed external libraries, we refine types as needed at the public interface of libraries using *annotation stubs*: files consisting of annotated type signatures for library methods. These annotations allow the typechecker to specify refinement types for third-party library code without checking the internals of those libraries or rebuilding them from source.

Annotation stubs are particularly effective for specifying the effects of Android library methods and the threading behavior of Observable operators. We present several such annotation stubs in Figure 7 in order to build intuition and demonstrate the technique.

- `ScrollView` is an Android UI element, so most of its methods have UI effect. The `@UIType` class-level shorthand applies `@UIEffect` to each of its methods; however, some methods – including `post` – can safely be called from non-UI threads and are thus annotated `@SafeEffect`.
- `delay` schedules the returned `Observable` on the computation threadpool, so its return value is annotated `@CompThread`.
- `observeOn` returns an `Observable` that emits events on the given `Scheduler`. This behavior is expressed by applying the polymorphic `@PolyThread` qualifier to both the return type and the `thread` parameter, constraining them to both have the same thread refinement.
- `take` simply truncates a stream after `k` events and does not affect the thread of the stream it is called on, as is the case with most `Observable` operators. We therefore constrain its receiverand return value to run on the same thread, using the polymorphic `@PolyThread` qualifier.

Annotation stubs are a possible source of unsoundness in our approach, since the refinement types they provide are trusted and the corresponding source code is unchecked. However, we are able to mitigate this concern through careful review of source code and documentation for both Android and ReactiveX. In particular, our annotation stubs conform to the Android developer guide's admonition that the `android.widget` and `android.view` packages comprise the Android UI toolkit (i.e. have UI effect) and to all threading behaviors specified in the ReactiveX `Observable` documentation [25, 50].

5 EVALUATION

We answer the following research questions in order to evaluate our approach.

(1) *Is the typechecker usable and practical?* Is the annotation burden sufficiently small for real-world Android developers to make use of the tool, and are the messages and warnings emitted by the tool useful and understandable?
(2) *Does the typechecker find real bugs and help fix them?* Are threading bugs in stream processing code prevalent in practice, does our tool identify them successfully, and is a program that successfully typechecks reliably free of such bugs?

5.1 Evaluation Suite

Our experimental evaluation consists of case studies over 8 open-source Android applications, as well as a report on our experience applying the tool in production to two Android applications developed at Uber

We select open-source applications for our experiments from Github according to the following criteria. First, we restrict our search to Android applications written in Java (excluding Android applications written in other languages since our typechecker operates over a Java AST).

Of those, we consider only applications that import a 2.x version of ReactiveX. We exclude applications using a 1.x version because our annotation stub coverage thereof is more sparse and applications using the older library version are more likely to be abandoned or broken.

Finally, we took the 8 most recently indexed applications with 10 or more Github "stars." These two criteria form an imperfect but practical proxy for repository activity: taking recently indexed apps avoids those that are abandoned or unmaintained, while requiring at least 10 stars ensures that the apps are not small personal projects or one-off experimental applications.

We believe that the 8 subject programs selected thusly form a reasonably representative cross-section of open-source Android projects, including, for example: a widely-forked template for Model-View-Presenter apps [61], a client for a Russian technology news website [45], and an app that scans, tracks, and organizes receipts [7]. In total, this open-source evaluation suite consists of 142 thousand lines of code written by 82 distinct contributors.

In addition to this corpus of open-source applications, we have also applied the typechecker to the Uber Eats and Uber Driver apps, both of which are large closed-source Android applications written and maintained by professional developers at Uber.

5.2 Annotation Workflow

This section details the process of applying the typechecker to an existing Android application.

First, we clone the application, and confirm that it builds successfully in our local environment. In practice, we found that all of the open-source apps gathered according to our criteria use the Gradle build system and were relatively easy to build locally [29]. Next, we configure the application's build to invoke our custom typechecker simply by adding package dependencies and setting `javac` command-line options in the `build.gradle` configuration file. At this point, the typechecker is fully configured and will be seamlessly integrated with the existing compilation process.

However, the task of refining effect and thread types of existing code remains. We find it most efficient to annotate in two phases:

```
@UIType class ScrollView {//...
  @SafeEffect boolean post(@UI Runnable action);}
class Observable<T> {//...
  @CompThread Observable<T> delay(long delay, TimeUnit unit);
  @PolyThread Observable<T> observeOn(@PolyThread Scheduler thread);
  @PolyThread Observable<T> take(@PolyThread Observable<T> this, int k);}
```

Figure 7: Selected example stub annotations for ReactiveX Observable methods and an Android UI class.

first, apply effect annotations throughout the application, ignoring any errors related to threading and then, once all methods have the proper effect type, apply thread annotations and/or fix real bugs until no alarms remain.

Recall the two conditions that govern effect typing: methods may only call methods with lesser effect (*transitivity*) and override methods with greater effect (*inheritance*).

Since all unannotated application code has `@SafeEffect` by default and all Android UI methods have `@UIEffect` from annotation stubs, every call from the application into the Android UI toolkit violates the transitivity condition in the app's initial (unannotated) state. In order to deal with this avalanche of alarms, we first identify all *packages* with names connoting UI effect (e.g., `ui`, `view`, `widget`, `layout`, etc.) and use the package-level `@UIPackage` annotation to apply `@UIEffect` to all methods therein.

Then, we triage remaining alarms and write type annotations manually until no alarms remain. We proceed from files with the most alarms to those with fewest, using the class-level `@UIType` for especially alarm-ridden classes where most methods have UI effect.

It is possible that our package- and class-level annotations assign the `@UIEffect` refinement to methods that do not actually interact with the UI. As such, manual triage is not as simple as blindly propagating `@UIEffect` annotations: we are careful to identify cases where an alarm is due to the package- and class-level annotations and use `@SafeEffect` to exclude a method from the coarse UI annotation rather than further propagating the spurious `@UIEffect`.

Unlike the effect type system, which requires a moderate amount of manual annotation, the thread type system requires almost none. Rather, our library annotation stubs are sufficient for the typechecker to determine the thread type refinement of intermediate values of fluent call-chains in most cases, and it is rare for developers to store `Observable` streams in instance fields or local variables, which would require type annotation at their declaration.

Therefore, once effect annotations are completed, the only remaining alarms in most applications are genuine threading violations, where a UI-effectful callback is subscribed to a non-UI-threaded stream. We manually inspect these violations to confirm that they are bugs and then find fixes on a case-by-case basis.

Once fixes have been found and applied, the application is verifiably safe from non-UI thread access of UI elements in stream-based code.

Crucially, this fairly involved annotation process is a one-time cost. After an application has been configured to use the thread and effect typechecker and fully annotated once, future bug fixes and feature additions require minimal annotations. Nonetheless, possible threading errors are automatically reported by the compiler as they are written, allowing developers to write and modify complex asynchronous stream-based software with confidence.

5.3 Experimental Results

For our experiments, we apply the described workflow to the 8 open-source applications gathered according to the aforementioned criteria. Figure 8 reports a summary of our results, which we apply here to the two stated research questions:

Is the typechecker usable and practical?

There are two facets to this question: first, is the annotation burden reasonable, both in terms of developer time and total annotation, and second, is the typechecker usable and understandable for real Android developers?

Our results on the open source evaluation suite demonstrate that, while the initial overhead of annotation is meaningful, it is far from prohibitive: we spent an average of 2.3 hours per application, writing one annotation per 186 source lines of code.

Does the typechecker find real bugs and help fix them?.

We find a total of 33 threading defects spread across 6 different apps in the open source evaluation suite, an average of 4.1 errors per application. The errors vary from simple oversights (e.g., forgetting an `observeOn`) to more complex interactions between multiple stream operators and combinators. The wide range of bugs found across a majority of subject programs demonstrates the applicability and efficacy of our approach.

We reproduce one example defect identified by the typechecker here, taken from the `ForPDA` [45] application and slightly modified for clarity.

```
observable.onErrorReturn(throwable -> {
  handleError(throwable,onErrorAction);
  return fallbackValue; })
.observeOn(AndroidSchedulers.mainThread())
.subscribe(callback)
```

The `onErrorReturn` method takes a lambda that runs whenever the receiver `Observable` emits an error, catching the error and emitting the lambda's return value instead.

In this case, the lambda simply calls `handleError` on the error `throwable` and returns a default value `fallbackValue`, before scheduling the resulting stream on the main thread and subscribing a callback `callback`. The `handleError` function renders the error message in the UI, but it may run on a background thread since `onErrorReturn` precedes the `observeOn` that moves the stream to the main thread. The fix to this bug is simple: switch the positions of `onErrorReturn` and `observeOn`, and the code compiles without error.

5.4 Uber Case Study

As part of our evaluation, we deployed the typechecker in Uber's development infrastructure for two major applications: the Uber Eats Android app and the newest version of the Uber Driver Android

App	KLoC	Annotations	Time Spent (hrs.)	Errors Found	Compile Time (sec.)
`ForPDA` [45]	33.0	197	3	4	27
`chat-sdk-android` [8]	34.6	102	2	6	21
`trust-wallet-android` [55]	8.8	27	1	2	17
`arch-components-date` [21]	0.7	2	0.5	0	8
`MVPArms` [61]	6.3	59	1	1	9
`rxbus` [37]	3.3	12	1	0	3
`SmartReceiptsLibrary` [7]	39.9	217	7	16	30
`OpenFoodFacts` [46]	14.9	146	3	4	41
Averages	17.7	95	2.3	4.1	19.5

Figure 8: Open-source test corpus experimental results. Reported LoC figures are computed with `sloccount` [59]. Reported compilation times are the mean of five executions of `gradle` compiling all application release sources with our typechecker enabled, measured on a laptop with an Intel i7-6700HQ processor and 8GB RAM, running Ubuntu 16.04.

app (then in development, it was released to partners in April 2018). The typechecker runs as part of the continuous integration pipeline in parallel with a manual code review process. It inspects every patch, blocking code from merging into the trunk of the repository if it fails to typecheck.

Android apps at Uber are organized as a collection of targets, compiled and unit tested separately. Over a period of 8 months, we progressively enrolled individual targets from each of the two apps onto the typechecker. For each target, we first wrote an initial set of annotations for existing code, then enabled typechecking for that target and merged in the new annotations simultaneously.

Overall, we enrolled over half a million LoC corresponding to targets from each of the two apps. As of this writing, we did not enroll targets in other Uber apps, or shared platform code and first-party libraries, although we did add (trusted but unchecked) annotations stubs for shared UI code.

During the initial enrollment process, we made 41 substantive changes to the application in addition to the added type annotations. Most of those changes are simple additions of `observeOn` calls to ReactiveX fluent call chains to move streams onto the main thread. These represent potential preexisting issues regarding improper UI access off the main thread. Although it is possible for many of them to be safe in practice, due to implicit knowledge about runtime behavior not captured in our type system, we consider the safety offered by the typechecker to be worth the potential marginal performance cost of moving these streams to the main thread or performing redundant `observeOn` calls.

During this enrollment period, we observed two critical production bugs involving ReactiveX streams accessing the UI from a background thread. The first was distilled into the `delay()` example shown in Figure 1. The second was an issue where developers failed to notice that `Observable#switchMap` could create and return a new `Observable` on a different thread, rather than propagate the thread of its receiver. These bugs were patched as soon as they manifested in production and the app updated accordingly, but both would have been caught by the current version of the typechecker had it already been enabled for the corresponding targets. This gives us some confidence that the tool is catching other similar issues before they make it past the development stage. By design, issues identified by the typechecker are fixed before they become production bugs.

After enrollment, any further commits changing files inside the corresponding target are checked against our type system and developers are responsible for maintaining the annotations as part of the development cycle. Over the 8 months of our rollout, over 4,000 commits and 178 developers interacted with our typechecker. We note that enrolled targets contain an average of one type annotation per 104 LoC, indicating that the burden placed on developers is relatively low.

Furthermore, when excluding commits made by the authors during the initial enrollment, the annotation burden falls to one type annotation per 179 LoC, implying that much of the annotation burden is incurred upfront and not as ongoing maintenance cost.

We measured that developers have added 135 `observeOn(...)` invocations to commits under code review in response to typechecker warnings, each representing a potential fix to a threading defect that could otherwise have gone uncaught. Note that we cannot verify that all of these changes are fixing real threading bugs; some may have been needed due to a spurious `@UIEffect` annotation or other typechecking imprecision. But, as we have not received developer feedback indicating excessive false positives, we believe that many of these changes were either fixing real issues or improving the code by making it more obviously safe. Catching these threading issues early in the development process reduces costs and improves productivity, as developers do not need to context-switch back to code they wrote days or weeks earlier to fix bugs.

5.5 Threats to Validity

The evaluation results presented in this section are predicated upon the correctness and soundness of our technique and experimental design.

First, it is important to note that our technique does not make any global guarantee about UI access or general thread-safety: the typechecker only verifies that UI-effectful code is annotated accordingly and stream-based code is safe from invalid UI thread access. As such, our tool does *not* find UI threading bugs outside of stream-based code or non-UI threading bugs such as deadlocks or data races.

Furthermore, the safety guarantees provided by the typechecker are sound with respect to trusted library annotation stubs, which could potentially diverge from the true behavior of the underlying code. We note, however, that our `@UIEffect` annotations for the

Android standard library include all methods explicitly specified as UI-effectful by the official developer guide [25].

Since these experiments were performed by an author of the tool, it is possible that the reported time spent underestimates the time that would be spent by a less experienced user. However, that factor is balanced somewhat by the fact that we were unfamiliar with the open-source applications being annotated – several of which were even documented in languages not spoken by the annotator.

It is also possible that our subject program corpus is not representative of stream-based Android programs in general, either due to small sample size or biases in our selection criteria.

6 RELATED WORK

Although there is a wealth of research on analysis of concurrent programs, existing approaches are ill-suited to the problem of detecting improperly threaded UI interactions in asynchronous stream-based software.

Much of the existing literature focuses on race detection, ordering violations, and deadlocks, leveraging a wide range of dynamic and static techniques including instrumentation, lockset algorithms, and model checking [20, 32, 44, 53].

However, the single thread model we wish to verify precludes the need for general thread analyses: there are no data races or deadlocks in a UI library that runs on only one thread. Furthermore, the aforementioned analyses are computationally expensive, relying on various pointer and alias analyses to reason about Java's heap-allocated threading constructs.

Our type-based approach builds upon a deep body of work in extensible type systems to address both of these issues with concurrency analysis, since typechecking is an intraprocedural analysis that scales well and our type system is designed specifically to identify improper UI thread access.

The idea of augmenting an existing type system with more expressive domain-specific types was introduced by Freeman and Pfenning in their seminal work refining algebraic datatypes in Standard ML [22]. However, their contributions are primarily theoretical, providing only a barebones proof-of-concept implementation.

Extending a language's syntax with *annotations* to express refinements, as originally proposed for Standard ML in [13], has emerged as a technique enabling practical adoption of refinement type systems. Annotation-based type systems are widespread, augmenting an existing static type system in Haskell [56] and SML [15] or adding simple static types to dynamically typed languages such as JavaScript [42] and Python [57].

In Java, the Checker Framework [48] provides a means to extend the Java base type system with arbitrary custom type refinements. This has also been used in the past, for example, to infer and check locking disciplines for multithreaded programs, verify information flow properties, and implement ownership and universe types [17, 18, 34].

The most closely related work to our own of which we are aware is the effect type system of Gordon et al. [28], which is also implemented using the Checker Framework. However, without the qualifier inference and thread typing techniques detailed in this paper, their typechecker is unable to verify the stream-based applications with which we are concerned.

Testing is a popular alternative to static analysis (type-based or otherwise) when trying to rule out classes of errors in software. There is a large body of work on automatically testing Android applications at the level of the UI. Fully automated testing approaches include random and search-directed event generation tools [12, 27, 35, 38, 40, 52, 63], model-based exploration [1, 2, 6, 10, 31, 43, 49, 62], concolic testing [3], and event generation using evolutionary algorithms [39].

Industrial state of the art mostly uses scripted UI tests constructed for a particular app using a testing framework [26, 51, 54, 60]. Record-and-replay systems also see widespread use to isolate bugs encountered during manual testing or after deployment [24, 30, 33] and can be combined with search-based approaches into hybrid testing schemes [41].

However, by their very nature, testing techniques cannot show the absence of bugs, only their presence. Even a theoretical perfect tester which explores all possible UI interaction traces can miss threading bugs. For example, issues flagged by our typechecker may depend on interaction with the network causing an event to be emitted on a stream that is subscribed by UI affecting code, but it is possible to explore every reachable view without this event ever occurring and thus miss the bug.

A survey by Choudhary et al. [11] suggests that existing fully automated UI testing tools do not significantly outperform random exploration in terms of code coverage under a fixed time budget. This could be due to either a faster rate of event generation for the simpler approach or the difference in engineering efforts going into an industry standard tool versus research prototypes. In either case, these approaches are challenging to scale for large and complex applications.

Finally, testing-based tools usually run late in the development process, whereas our analysis can run before a code change has even been merged into the trunk of the repository for our apps.

7 CONCLUSION

We present in this paper a technique that statically verifies UI access to occur only from the UI thread in stream-based Android programs. Our approach refines the types of data streams with a static bound on their thread, refines the types of methods and callback-like objects with a static bound on their side-effects, and collates the two type systems in order to detect invalid thread accesses or prove the absence thereof.

We demonstrate the efficacy and usefulness of our typechecker by evaluating it on 8 open-source applications and finding 33 bugs. We also report on our experience applying the tool at scale to two large production Android applications at Uber, where it has analyzed over 4000 commits by 178 developers at time of writing.

ACKNOWLEDGMENT

The authors would like to thank Werner Dietl, Colin Gordon, Michael Ernst, and members of the CUPLV lab for valuable discussions.

This material is based on research sponsored in part by the National Science Foundation under grant number CCF-1055066 and by DARPA under agreement number FA8750-14-2-0263. The U.S. Goverernment is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 258–261.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59.

[3] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 59.

[4] Apple. 2018. *App Programming Guide for iOS.* https://developer.apple.com/library/content/documentation/iPhoneOSProgrammingGuide.

[5] Apple. 2018. *MacOS Cocoa.* http://developer.apple.com/technologies/mac/cocoa.html.

[6] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 641–660.

[7] Will Baumann. 2018. *Smart Receipts.* https://github.com/wbaumann/SmartReceiptsLibrary.

[8] Chat SDK. 2018. *Chat SDK Android.* https://github.com/chat-sdk/chat-sdk-android.

[9] Checker Framework Developers. 2018. *Checker Framework Manual.* https://checkerframework.org/manual.

[10] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 623–640.

[11] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 429–440.

[12] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. 2016. Minimizing GUI Event Traces. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 422–434.

[13] Rowan Davies. 1997. Refinement-Type Checker for Standard ML. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*. 565–566.

[14] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd W. Schiller. 2011. Building and Using Pluggable Type-checkers. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 681–690.

[15] Joshua Dunfield. 2007. Refined Typechecking with Stardust. In *Proceedings of the ACM Workshop Programming Languages meets Program Verification (PLPV)*. 21–32.

[16] Eclipse. 2018. *Standard Widget Toolkit.* http://eclipse.org/swt.

[17] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward XueJun Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1092–1104.

[18] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. 2016. Locking Discipline Inference and Checking. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1133–1144.

[19] Mattia Fazzini and Alessandro Orso. 2017. Automated Cross-Platform Inconsistency Detection for Mobile Apps. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) 2017*. 308–318.

[20] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[21] Rebecca Franks. 2018. *Date Countdown.* https://github.com/riggaroo/android-arch-components-date-countdown.

[22] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 268–277.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[24] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd D. Millstein. 2013. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 72–81.

[25] Google. 2018. *Android Developer Guide.* https://developer.android.com/guide.

[26] Google. 2018. *Espresso.* https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[27] Google. 2018. *UI/Application Exerciser Monkey.* https://developer.android.com/tools/help/monkey.html.

[28] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. 2013. Java UI : Effects for Controlling UI Object Access. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 179–204.

[29] Gradle, Inc. 2018. *Gradle Build Tool.* https://gradle.org.

[30] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: Cross-Platform User-Interaction Record and Replay for the Fragmented Android Ecosystem. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 215–224.

[31] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-Automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 204–217.

[32] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Race Checking by Context Inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1–13.

[33] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet Lightweight Record-and-Replay for Android. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 349–366.

[34] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. 2012. Inference and Checking of Object Ownership. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 181–206.

[35] Bo Jiang, Yuxuan Wu, Teng Li, and W. K. Chan. 2017. SimplyDroid: Efficient Event Sequence Simplification for Android Application. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 297–307.

[36] JSR 308 Expert Group. 2014. *Annotations on Java Types.* http://download.oracle.com/otndocs/jcp/annotations-2014_01_08-pfd-spec.

[37] Kuwork. 2018. *RxBus.* https://github.com/kuwork/rxbus.

[38] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE/ESEC)*. 224–234.

[39] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 599–609.

[40] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

[41] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd Intelligence Enhances Automated Mobile Testing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 16–26.

[42] Microsoft. 2018. *TypeScript.* https://www.typescriptlang.org.

[43] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 559–570.

[44] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 308–319.

[45] Evgeny Nizamiev. 2018. *ForPDA.* https://github.com/RadiationX/ForPDA.

[46] Open Food Facts, Org. 2018. *Open Food Facts.* https://github.com/openfoodfacts/openfoodfacts-androidapp.

[47] Oracle. 2018. *JDK Swing Framework.* http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[48] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 201–212.

[49] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*. 209–220.

[50] Reactive Extensions. 2018. *Reactive Extensions.* reactivex.io.

[51] Robotium. 2018. *Robotium.* https://github.com/robotiumtech/robotium.

[52] Raimondas Sasnauskas and John Regehr. 2014. Intent Fuzzer: Crafting Intents of Death. In *Proceedings of the Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. 1–5.

[53] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*. 27–37.

[54] Selendroid. 2018. *Selendroid.* http://selendroid.io/.

[55] Trust. 2018. *Trust Wallet.* https://github.com/TrustWallet/trust-wallet-android-source.

[56] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Proceedings of the European Symposium on Programming (ESOP)*. 209–228.

[57] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the ACM*

Symposium on Dynamic Languages (DLS). 45–56.
[58] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 226–237.
[59] David A. Wheeler. 2004. *SLOCCount*. https://www.dwheeler.com/sloccount/.
[60] Xamarin. 2018. *Calabash*. http://calaba.sh/.
[61] Jess Yan. 2018. *MVP Arms*. https://github.com/JessYanCoding/MVPArms.
[62] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*. 250–265.
[63] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of the International Conference on Advances in Mobile Computing & Multimedia (MoMM)*. 68.
[64] Sai Zhang, Hao Lü, and Michael D. Ernst. 2012. Finding Errors in Multithreaded GUI Applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 243–253.