# nAdroid: Statically Detecting Ordering Violations in Android Applications

Xinwei Fu
Virginia Tech, USA
fuxinwei@vt.edu

Dongyoon Lee
Virginia Tech, USA
dongyoon@vt.edu

Changhee Jung
Virginia Tech, USA
chjung@vt.edu

## Abstract

Modern mobile applications use a hybrid concurrency model. In this model, events are handled sequentially by event loop(s), and long-running tasks are offloaded to other threads. Concurrency errors in this hybrid concurrency model can take multiple forms: traditional atomicity and ordering violations between threads, as well as ordering violations between event callbacks on a single event loop.

This paper presents nAdroid, a static ordering violation detector for Android applications. Using our threadification technique, nAdroid statically models event callbacks as threads. Threadification converts ordering violations between event callbacks into ordering violations between threads, after which state-of-the-art thread-based race detection tools can be applied. nAdroid then applies a combination of sound and unsound filters, based on the Android concurrency model and its happens-before relation, to prune out false and benign warnings.

We evaluated nAdroid with 27 open source Android applications. Experimental results show that nAdroid detects 88 (at least 58 new) harmful ordering violations, and outperforms the state-of-the-art static technique with fewer false negatives and false positives.

***CCS Concepts*** • **Software and its engineering** → **Software testing and debugging**;

***Keywords*** Ordering violation, Data race, Use-after-free, Static analysis, Debugging, Android, Threadification

**ACM Reference Format:**
Xinwei Fu, Dongyoon Lee, and Changhee Jung. 2018. nAdroid: Statically Detecting Ordering Violations in Android Applications. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3168829

## 1 Introduction

The mainstream mobile platforms (Android, iOS, Windows Phone) offer a distinctive concurrent programming model. Mobile applications are often sensor-driven (e.g. touchscreen, GPS), and sensor data is most readily handled by the event-driven model. On the other hand, developers want to take advantage of the multiprocessors in modern mobile devices [32]. To accommodate these competing demands, the concurrency model on mobile platforms is a hybrid of event loop(s) that sequentially handle events, and background threads that concurrently execute long-running tasks.

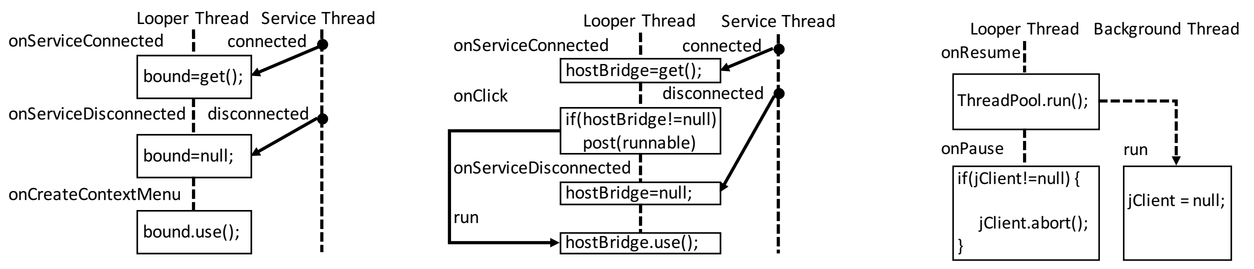
While this hybrid model allows developers to balance responsiveness with performance, it also exposes mobile applications to new classes of concurrency errors. While mobile applications can of course contain conventional *multi-threaded data races* due to a non-deterministic thread schedule, recent studies have shown that these applications can also contain *single-thread data races* resulting from a non-deterministic event posting order [3, 9, 17, 26]. These concurrency errors have been shown to cause issues like performance degradation [24], unexpected termination [3], accelerated battery drain [33], and security vulnerabilities [6–8].

Concurrency errors in mobile systems have been tackled with both dynamic and static techniques. Several works test mobile applications dynamically, collecting execution traces and performing offline data race detection [3, 17, 26]. Though dynamic testing has relatively few false positives, the detection coverage is limited to the observed executions.

In contrast, static analysis can inspect program code for all possible runtime behaviors. However, the use of static program analysis to detect concurrency errors has not yet been well studied in the context of mobile applications. Conventional static data race detectors [11, 12, 19, 29, 35, 43] only focus on multi-threading, making them unsuitable for event-driven mobile applications. Recently, several new techniques have been proposed for mobile applications [24, 33, 39], but their scope is limited: e.g., they do not consider the happens-before relationship between event callbacks (§2).

This paper presents nAdroid[1], a novel static ordering violation detector for mobile applications with hybrid concurrency model, tailored to Android. Though nAdroid can statically detect all types of data races, classifying data races

[1]nAdroid is named after Android, but has an "ordering violation" between the first two letters.



**(a)** single-threaded UAF 1 in `ConnectBot` **(b)** single-threaded UAF 2 in `ConnectBot` **(c)** multi-threaded UAF in `FireFox`

**Figure 1.** Examples of harmful use-after-free (UAF) ordering violations.

as *harmful* or *benign* is hard in general [20, 30, 45]. Therefore, this study focuses on finding *use-after-free* (UAF) ordering violations (e.g., `f=null` vs. `f.use()`). A UAF ordering violation is a form of *harmful* read-after-write data race, because it can lead to an unexpected `NullPointerException`.

The key challenge is that the event-based and thread-based programming models have distinct patterns and dissimilar happens-before relations [22], making it hard to statically detect them together. nAdroid addresses this problem using our novel *threadification* technique (§4) that statically models the event-driven aspects of Android applications as threads. In effect, threadification converts the tricky problem of detecting single-threaded ordering violations between event callbacks into the well-studied problem of detecting multi-threaded ordering violations. This allows nAdroid to leverage existing static data race detectors developed for multi-threaded programs (Chord [29] in our study) to detect both event-driven and threaded ordering violations in a unified manner (§5).

Furthermore, nAdroid introduces novel static *happens-before-based filters*, crafted based on the Android concurrency model, to prune out false UAF warnings. It is critical to remove false positives as they are often overwhelming enough to make programmers unwilling to use a detection tool. The problems specific to the Android context have not been addressed by existing static tools. We describe sound and unsound filters for these problems (§6).

We evaluated nAdroid using 27 open-source Android applications, from which nAdroid detects 88 (at least 58 novel) true harmful ordering violations. Experimental results also show that nAdroid produces fewer false negatives and fewer false positives than the state-of-the-art static technique, DEvA [39].

This paper makes the following contributions:

- We present nAdroid, a static ordering violation detector for Android, that considers both the event-based and the thread-based concurrency models. We model event callbacks as threads so that ordering violations between callbacks and threads can be detected in a unified manner.
- To the best of our knowledge, nAdroid is the first tool that incorporates the Android concurrency model and happens-before relations into static analysis to prune false or benign warnings.
- We evaluate 27 Android applications and show that nAdroid detects true harmful ordering violations, and produces fewer false positives and false negatives than the state-of-the-art static technique.

## 2 Background and Motivation

This section introduces the Java-based Android concurrency model, provides three examples of harmful UAF ordering violation that nAdroid found, and demonstrates the limitations of existing techniques.

### 2.1 The Android Concurrency Model

An Android application has a hybrid event-driven and thread-based concurrency model to handle a mix of incoming sensor data and user interactions (UI) best addressed with events, and to support arbitrary processing, best addressed with threads. A thread may attach an *event queue* to itself and handle an *event* for execution. A thread with an event queue is called a *looper thread*. It continuously checks its event queue and processes one event at a time by executing the corresponding *event callback*. Therefore, all the event callbacks executed in one looper thread are atomic (no preemption) with respect to each other. Furthermore, the application may create additional native threads. Since it is cumbersome for a native thread to communicate with a looper thread, the Android framework also provides a high-level concurrent construct, `AsyncTask`, to create a child thread that can interact with the looper thread via events.

### 2.2 Examples of UAF Violations

Figure 1 shows some real examples of *harmful* UAF ordering violations that nAdroid found in Android applications. The first two cases (a) and (b) are from `ConnectBot`, and represent UAF bugs between event callbacks within a single looper thread. In this example, `ConnectBot` binds to a background service. Its `onServiceConnected` and `onServiceDisconnected` callbacks are invoked by the framework to allow it to interact

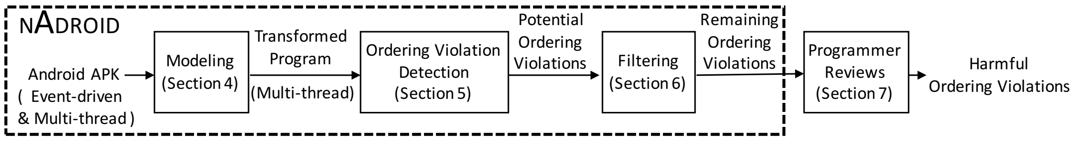


**Figure 2.** nAdroid analyzes an APK package and reports potential ordering violations between callbacks and threads.

with the service. In case (a), bound is used by the onCreateContextMenu callback without ensuring that the service remains connected. If onServiceDisconnected runs before onCreateContextMenu, then the program crashes with an exception. In case (b), onClick actually checks if the field hostBridge is not null. Unfortunately, it then posts a Runnable to the looper thread that will use the hostBridge later, asynchronously. If onServiceDisconnected is triggered before the Runnable, then hostBridge will be set to null, and the Runnable will throw a NullPointerException. In cases (a) and (b), the ConnectBot developers were clearly aware of the possibility of service disconnection – after all, they implemented onServiceDisconnected – but they were incautious about potential timing issues.

Case (c) from FireFox shows a UAF bug between a looper thread and a background thread. onResume submits a Runnable object to the thread pool, which sets jClient to null. Though onPause checks if jClient is null, it is prone to a UAF error due to the lack of atomicity.

### 2.3 Limitations of Current Techniques

Existing trace-based dynamic data race detectors [3, 17, 26] suffer from a coverage problem. They collect execution traces from manual [17] or automatic UI exploration [3, 26], and then perform offline data race detection. Though these dynamic detectors soundly detect all races for observed traces, their coverage is limited by their input generator. For example, CAFA reports no harmful races between event callbacks in ConnectBot (see Table 1 in [17]), while nAdroid found 13.

For static analysis, the state of the art is DEvA [39], which performs whole-program static ordering violation ("event anomaly") analysis for Android applications. However, DEvA is limited by significant sources of both false positives and false negatives. Notably, DEvA does not adequately consider the happens before relationship between event callbacks, resulting in many false positives. nAdroid's happens-before based filters (§6) specifically address this problem, saving developers' manual debugging efforts.

In addition, DEvA suffers from two sources of false negatives. First, DEvA does not take into consideration the multi-threaded aspect of the Android programming model, and applies its if-guard and intra-allocation filters (§6) unsoundly, assuming that all methods are executed atomically regardless of whether they are invoked concurrently. Second, DEvA's read/write set analysis is intra-class, restricted to within each class and its inner classes, i.e., DEvA cannot identify any inter-class racy accesses. Thus, DEvA ignores the common Android programming practice of putting event callbacks and associated background threads in separate classes (e.g. Runnable, Handler and AsyncTask classes), missing ordering violations that frequently occur across different classes.

For other static tools, Asynchronizer [24] handles only one particular form of thread, namely the AsyncTask; and Pathak et al. [33] consider only the fork-join relationship.

## 3 Overview

Figure 2 shows an overview of nAdroid. Given an Android APK package, nAdroid reports potential UAF violations.

An Android program is a collection of event callbacks which are asynchronously invoked by the Android framework based on UI, sensor input, Android component lifecycles, and so on. Without a single entry point, static analyses do not know where to begin. Like previous work [1, 4], nAdroid introduces a "dummy main" to create a single entry point so that the resulting program can be consumed by existing analysis frameworks [21, 29, 44].

nAdroid's dummy main is unique as it *models event callbacks as threads*, which we call *threadification* (§4). nAdroid first analyzes the APK package to identify the application's different entry points (event callbacks). nAdroid, at a high level, adds one child thread to the dummy main for each entry point. nAdroid applies this analysis recursively, repeatedly modeling the callbacks registered and threads created by these child threads as new child threads.

Given this transformed program, nAdroid can then apply a static data race detector developed for (conventional) multi-threaded programs to detect data races in Android applications. In this study, nAdroid uses Chord [29], the state-of-the art open source static data race detector for Java programs with some Android-specific modifications (§5) to detect potential ordering violations.

nAdroid prunes out many false and benign warnings using a set of filters based on the Android concurrency model and its happens-before relation, as well as on common Android programming patterns (§6).

Finally, nAdroid gives programmers insight into the remaining ordering violation warnings so that they can determine whether they are truly harmful at runtime (§7).

## 4 Threadification

Our key insight is that, through threadification, we can convert *single-threaded ordering violations* between unordered

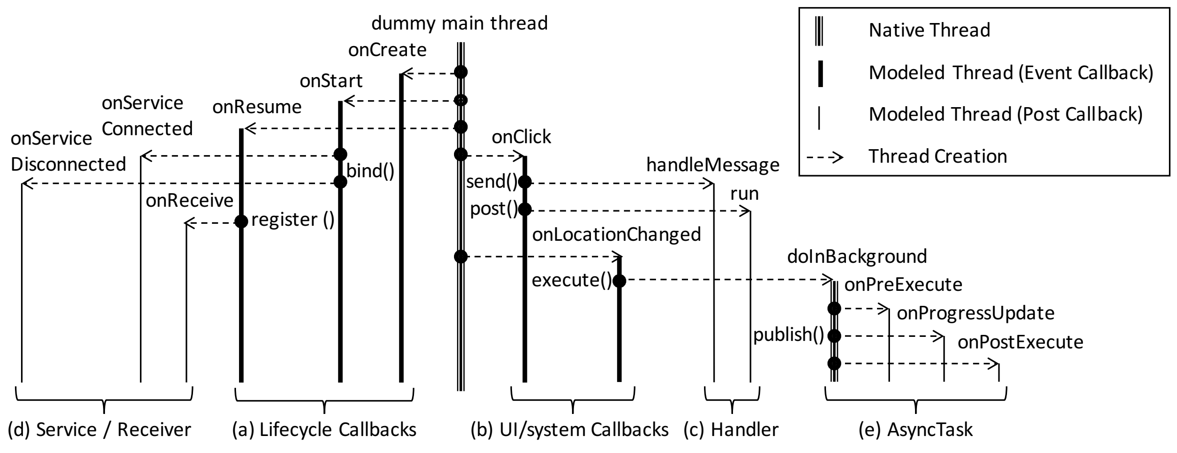


**Figure 3.** Through our threadification technique, an event-driven program is transformed into a multi-threaded program.

event callbacks into *multi-threaded ordering violations* between threads. After threadification, we can apply an ordering violation detection algorithm. This approach allows nAdroid to incorporate both event callbacks and threads in the analysis scope. The ability to re-use existing static analysis tools is an ample reward (§5).

Figure 3 shows an example of a transformed program. Threadification creates a dummy main thread (representing the initial looper thread) that creates "threads" in response to various application behaviors. At a high level, there are two forms of callbacks: Entry Callbacks (EC), externally invoked by the Android runtime, and Posted Callbacks (PC), internally triggered by other event callbacks or threads. nAdroid models ECs as child threads of the dummy main to mimic the way that Android lifecycle, UI, and other entry callbacks are executed by the initial looper thread. Then, nAdroid models PCs as child threads of the posting callbacks/threads to preserve the causal relation between the poster and the postee, reducing false positives. This lineage also helps programmers reason about the callback and thread sequence associated with each potential UAF bug (§7).

The proposed modeling does not capture precise happens-before orders between callbacks (e.g. lifecycle ECs), nor does it reflect the atomic execution of callbacks. In §6, we discuss introduce sound and unsound filters to mitigate these sources of imprecision. A detailed description of our transformation technique follows.

## 4.1 Entry Callbacks (EC)

Android programs use the `Activity` component to handle user interactions. The `Activity` component frequently transitions between different states in its lifecycle. For example, a running activity becomes paused when another activity comes into the foreground. For each transition, the Android framework calls a (pre-defined) lifecycle callback such as `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, and `onDestroy`. Because they are externally invoked by the Android runtime (not internally by other event callbacks), nAdroid models them as child threads spawned by the dummy main (UI) thread. Figure 3 (a) shows an example with `onCreate`, `onStart`, and `onResume`.

Furthermore, Android programs include other (non-lifecycle) event callbacks for coping with UI interactions (e.g., button clicks) and system events (e.g., GPS, accelerometers). These callbacks may be registered imperatively using Android framework APIs (e.g., a `requestLocationUpdates` method may register an `onLocationChanged` callback) or declared in the manifest XML files. Similar to the lifecycle case, these events are externally posted by the Android runtime. Thus, nAdroid models them as child threads of the dummy main thread. Figure 3 (b) shows how `onClick` and `onLocationChanged` callbacks are modeled.

## 4.2 Posted Callbacks (PC)

nAdroid models three forms of posted callbacks: `Handler`, `Service`/`BroadcastReceiver`, and `AsyncTask` callbacks.

First, Android's `Handler` provides two generic methods by which a callback (or a thread) may communicate with a looper thread. The `sendMessage` method delivers a `Message` object to be processed by the recipient's `handleMessage` method, while the `post` method enqueues a `Runnable` object to be executed later by the receiving looper thread[2]. Both of these methods post an event to the receiving looper thread. Since these events come from within the application itself, nAdroid models them as threads created by the caller of these methods. This approach allows nAdroid to establish a happens-before order between the caller's instructions preceding the call and the instructions performed by the newly-modeled thread, reducing false positives. Figure 3 (c)

[2]Android provides other methods to post a `Runnable` object, including `View`'s `post` and `Activity`'s `runOnUIThread` methods. nAdroid addresses these in the same fashion.

illustrates an example in which onClick calls both sendMessage and post; the resulting activities are modeled as child threads.

Second, Service and BroadcastReceiver are the two remaining major components in the Android model. The Service component handles background processing, and the BroadcastReceiver component responds to system-wide broadcast announcements. After an application binds to a background service (using bindService), its onServiceConnected and onServiceDisconnected callbacks may be invoked. Therefore, nAdroid models these callbacks as child threads of the caller of the bindService method. Similarly, an application may register a broadcast receiver (using registerReceiver). Then, the application's onReceive callback is executed on subsequent broadcast deliveries. nAdroid models this callback in the same manner. Figure 3 (d) presents an example in which Service and BroadcastReceiver are respectively bound and registered by onStart and onResume.

Third, to facilitate background processing, the Android framework provides a high-level concurrency construct, AsyncTask, by which a looper thread can create and easily communicate with a child thread. The AsyncTask can publish results using the publishProgress method, which will trigger the onProgressUpdate callback of its parent looper thread. In addition, the framework invokes the parent's onPreExecute callback before the AsyncTask begins, and its onPostExecute callback after it ends.

nAdroid preserves the parent-child relationship of any AsyncTasks in the application, and models the associated onPreExecute and onPostExecute callbacks as children of the AsyncTask. Figure 3 (e) shows an example in which onLocationChanged executes an AsyncTask (doInBackground), which in turn creates the three child threads shown.

The Android framework also allows a program to create native Java threads (java.lang.Thread). nAdroid simply treats these as threads.

## 5 Detecting UAF Ordering Violations

After threadification, nAdroid uses a modified version of Chord [29], the state-of-the-art static data race detector for Java programs, to detect UAF ordering violations in Android applications. Chord supports context-sensitive heap object naming and alias analysis, namely *k-object-sensitive-analysis* [27, 28]. Then, Chord performs static data race detection based on k-object-sensitive points-to analysis, thread escape analysis, lockset analysis, and May-Happen-in-Parallel (MHP) analysis. Modeling event callbacks as threads allows nAdroid to take advantage of such precise analyses developed for multi-threaded programs.

nAdroid makes three modifications to the original Chord algorithm to focus on UAF bugs. First, we filter out data races that cannot be a UAF violation. nAdroid defines a *use* as the Java bytecode getfield retrieving the field of an object, and

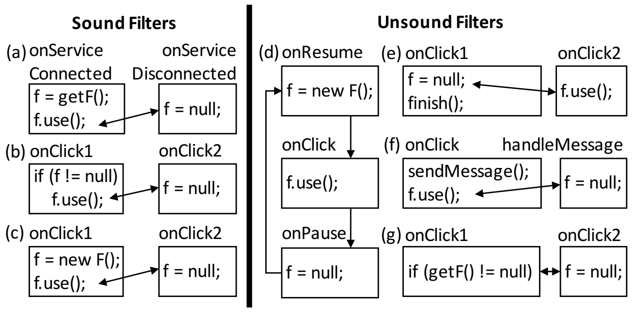


**Figure 4.** Examples of sound filters (a) MHB (b) IG (c) IA, and unsound filters (d) RHB (e) CHB (f) PHB (g) UR

a *free* as the Java bytecode putfield setting a field to null. nAdroid only considers racy pairs that consist of a *use* and a *free* on the same field.

Second, because locks do not prevent UAF errors, nAdroid initially ignores Chord's lockset analysis. Though locks provide atomicity, they cannot prevent ordering violations; UAF bugs can happen with or without locks. Therefore, nAdroid considers all racy *use*/*free* pairs no matter the locking pattern. Since locks may be used to prevent UAFs when combined with other checks, we refine our algorithm in §6.1.

Third, nAdroid does *not* use Chord's MHP analysis. For Android applications, the MHP analysis does not add much value because (blocking) synchronization primitives enforcing a specific order are not frequently used. Moreover, Chord's simple MHP analysis does not support sophisticated notify/wait analysis as in [23, 31]. Nonetheless, it requires flow-sensitive analysis which often suffers from scalability issues. Instead, we introduce Android-specific happens-before filters in the next section to replace MHP analysis designed for conventional multithreaded programs.

## 6 Filtering False Positives

To prune false UAF violations, nAdroid introduces novel static happens-before (HB)-based filters derived from the Android concurrency model, and non-HB filters that are derived from common Android programming patterns. Existing dynamic tools have defined happens-before relations in Android applications [3, 17, 26], but our use of the Android happens-before relation in static analysis is novel.

### 6.1 Sound Filters

nAdroid employs three sound filters: must-happen-before (MHB), if-guard (IG) and inter-allocation (IA).

#### 6.1.1 Must-Happens-Before (MHB)

In Android, some event callbacks have a *must-happens-before* (MHB) relation to each other; one must always happen before the other. When a potential UAF is detected but the use must precede the free, it is a false UAF. nAdroid identifies three statically sound MHB relations in Android, pruning UAF violations where they occur.

**MHB-Service:** When an `Activity` binds to a `Service`, the `onServiceConnected` callback is always executed before the `onServiceDisconnected` callback. Figure 4 (a) presents an example of a false UAF in this context.

**MHB-AsyncTask:** For `AsyncTask`, a looper thread's `onPreExecute` callback must happen before both the native thread's `doInBackground` method and the looper thread's `onProgressUpdate` callback; and all of these must happen before the looper thread's `onPostExecute` callback. nAdroid prunes MHB races in this context.

**MHB-Lifecycle:** `Activity` obeys the framework-defined lifecycle, which ensures that all the UI callbacks must happen after `onCreate` and before `onDestroy`. This introduces other MHB relations for which nAdroid accounts.

We emphasize that, statically speaking, there are *no* MHB relations for `onResume`, `onPause`, and other similar UI callbacks based on the Android lifecycle. The back edge from `onPause` to `onResume` (i.e. the "back button") introduces a circular dependency. In fact, harmful UAF transitions arising from use of the back button are common in Android applications. Consider the sequence in Figure 4 (d), and suppose for now that the `onResume` callback makes no allocation `f = new F();`. When `onPause` occurs, it sets `f = null`, and the activity becomes visible again. Then the user may trigger `onClick`, leading to a UAF violation.

#### 6.1.2 If-Guard (IG)

To prevent UAF errors, it is common practice to check whether a reference is null before using it. This "if-guard" is safe as long as atomicity between the check and the use is guaranteed and thus the free operation cannot be interleaved. This is true for event callbacks that belongs to the same looper thread. Thus, the IG filter prunes these false UAF warnings between event callbacks (modeled threads). Figure 4 (b) shows an example where `OnClick1` checks `f` before using it in an atomic callback.

On the other hand, an if-guard is unsafe in the presence of concurrency (thread-thread or event loop-thread) due to the lack of atomicity. Therefore, the IG filter removes UAF warnings between them only if they are protected by the same lock, providing atomicity. nAdroid uses Chord's lockset analysis selectively for this purpose. We note that previous dynamic tools [17] also use the IG and the following IA filters that require atomicity, but the static DEvA[39] tool applies them unsoundly without atomicity analysis, resulting in false negatives.

#### 6.1.3 Intra-Allocation (IA)

As shown in Figure 4 (c), if there is an allocation before use within an atomic event callback `onClick1`, free operations in other callbacks (e.g. `onClick2`) in the same looper thread cannot lead to UAF violations. The same is true for warnings between native thread protected by the same lock. The IA filter prunes this case using an intra-procedural data-flow analysis. If inter-procedural analysis is required to determine the location of the allocation (e.g., `getF()` in Figure 4 (a)), nAdroid conservatively does not filter out the case. The unsound Maybe-Allocation (MA) filter (§6.2) assumes that the getter method does not return null.

### 6.2 Unsound Filters

Without runtime information, a static analysis must conservatively apply filters, and using only sound filters leads to many false positives. Our unsound, yet effective filters are derived from studying the UAF warnings that survive our sound filters. We used seven of the applications evaluated in the CAFA study [17] as a training set. By applying our sound filters to each application and manually determining the false positives, we identified common *may-happens-before* relations and Android programming idioms. Part of our evaluation (§8) measures the effectiveness of these unsound filters and gives a detailed analysis on the resulting false negatives.

For users who demand soundness, these (optional) unsound filters serve as a ranking system that allows programmers to focus on the still-unpruned remaining races first as the most likely harmful UAF violations in their application.

#### 6.2.1 May-Happen-Before (mayHB)

nAdroid identifies three likely-true happens-before relations between event callbacks, and uses them as unsound filters.

**Resume-Happens-Before (RHB):** As discussed in §6.1, the MHB-Lifecycle filter is not applied to `onResume`, `onPause`, and other UI callbacks. Given that an `Activity` is often paused and resumed, careful programmers use the `onResume` callback to ensure program invariants and correctness across lifecycle transitions. Figure 4 (d) shows an example of this practice, using an allocation in `onResume` to prevent a UAF violation in `onClick`. To accommodate this pattern, the Resume-Happen-Before (RHB) filter prunes out UAF warnings between a UI event callback and `onPause` if there is a corresponding allocation in `onResume`. In a sense, this filter can be viewed as extending the IA filter (§6.1) across a callback boundary. The RHB filter is *unsound* because nAdroid performs *may* analysis for identifying allocations in `onResume`: if any program path in `onResume` allocates the object in question, the RHB filter assumes that the corresponding use operation is safe[3].

**Cancel-Happens-Before (CHB):** The use of Android's API-based cancellation methods establishes a cancel-happens-before (CHB) relation between the canceller and the affected callbacks. Android programs can close an `Activity` (via the `finish` method); unbind from a `Service` (`unbindService`); unregister from a `BroadcastReceiver` (`unregisterReceiver`); or remove any pending posts of callbacks and sent messages

[3]This can be dealt with by a path-sensitive analysis, but it is not scalable.

from a Handler (removeCallbacksAndMessages). For example, no UI callbacks will occur after an event callback invokes the finish method. The same reasoning affects service-, receiver-, and handler-related callbacks like onServiceConnected, onServiceDisconnected, and onReceive. In Figure 4 (e), we see a false UAF violation between onClick1 and onClick2 that can be pruned by the CHB filter. If onClick2 could happen after onClick1, there would be a UAF violation, but onClick2 must happen before onClick1 because onClick1 calls finish. Similar to RHB, the CHB filter is *unsound* because of path-insensitive may-analysis for scalability.

**Post-Happens-Before (PHB):** There is a post-happens-before (PHB) order between two callbacks on the same looper thread when one posts an event that triggers the other. Figure 4 (f) shows an example in which onClick uses sendMessage to send a message to its looper thread. As callbacks on the same looper thread are atomic, this causes the handleMessage callback to be scheduled only after the completion of onClick. The PHB filter prunes out a UAF violation warning when there is a post-based happens-before order between the use and free operations. The PHB filter is *unsound*: it assumes that two different instances of UI event callbacks do not share an object/field at runtime. If they do, in our example another call to the onClick callback (user clicks the button multiple times) may lead to a UAF error.

### 6.2.2 Maybe-Allocation (MA)

nAdroid applies the sound IA filter (§6.1) when an event callback allocates an object with new before using it. However, obtaining an object reference could instead be done with a getter method, as shown in Figure 4 (a). To avoid expensive inter-procedural analysis, the *unsound* maybe-allocation (MA) filter assumes that custom getter methods never return null, thus acting like the IA filter.

### 6.2.3 Used-for-Return (UR)

The used-for-return (UR) filter prunes out commonly-benign uses associated with getter methods and method invocations. When a getter method, say getF(), simply returns a reference, it leads to a use in our analysis. However, this reference is often used safely later for comparison, as shown in Figure 4 (g), making that and any subsequent use benign. The UR filter also prunes warnings associated with passing a field as a parameter to a method invocation, as these are often benign for a similar reason. Determining whether references are used safely in these cases would require expensive inter-procedural analysis.

### 6.2.4 Thread-Thread (TT)

Given that we are more interested in UAF bugs that are unique to Android programs including single-threaded races between callbacks, and multi-threaded races between event callbacks (in a looper thread) and native threads, the thread-thread (TT) filter prunes out the UAF warnings that occur between native threads and do not involve a looper thread. Detecting data races between two native threads has been well studied with conventional multi-threaded race detectors [12, 29, 35, 43].

## 7 Validating Harmful UAF bugs

The potential UAF bugs uncovered statically by nAdroid can be confirmed as harmful with a dynamic execution that triggers a NullPointerException. As constructing an execution to trigger a UAF bug is a project unto itself, nAdroid simply offers programmers aids to ease a manual analysis. The first step in these aids is to classify the UAF warnings.

nAdroid groups UAF warnings based on the origins of the use and free operations, based on the type of event callback and/or thread involved. nAdroid categorizes event callbacks (C) into Entry Callbacks (EC) and Posted Callbacks (PC). EC includes lifecycle, UI, and other system-triggered callbacks (Figure 3 (a) and (b)). Every other callback is a PC. We classify any native threads created by an event callback as Reachable Threads (RT) relative to this callback, and all other native threads as Non-Reachable Threads (NT) relative to it. For example, in Figure 3, doInBackground is reachable from onLocationChanged, but not from onClick. Thread reachability is transitive across thread creation and event posting.

With these terms in mind, nAdroid provides two aids to help programmers more rapidly analyze each potential UAF bug. First, nAdroid provides programmers with the callback and thread sequence associated with each potential UAF bug. For the example in Figure 3, suppose a UAF warning between the handleMessage and onProgressUpdate callbacks. Then, nAdroid can tell that the handleMessage callback is derived from onClick callback, and that the onProgressUpdate stems from the onLocationChanged callback via the doInBackground thread.

Second, nAdroid's potential UAF report includes whether event callbacks are ECs or PCs, and whether threads are RT or NT. This allows programmers to sort potential UAF bugs based on two hypotheses we provide about common sources of errors in the Android environment. Our intuition is that more complex interactions are more likely to be the source of UAF bugs. First, we suggest that true UAF bugs will occur more frequently between PC-EC or PC-PC than between EC-EC, because PCs are more asynchronous and difficult to reason about. Second, we believe C-NT are more prone to UAF bugs than C-RT because it is hard to reason about interactions between seemingly independent callbacks and threads. We provide some empirical support for these hypotheses in §8.4.

## 8 Evaluation

This section demonstrates our evaluation. Our evaluation answers the following questions:

- Are nAdroid's sound and unsound filters effective? (§8.3)
- Can nAdroid find harmful UAF ordering violations? (§8.4)
- What are the sources of false warnings? (§8.5)
- Does nAdroid have false negatives? (§8.6)
- How does nAdroid compare to DEvA? (§8.7)
- What is the overhead of nAdroid's static analysis? (§8.8)

### 8.1 Implementation and Subject Systems

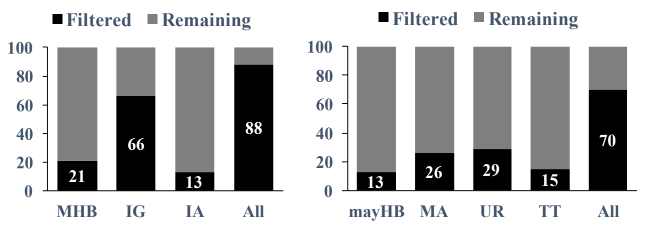
For threadification (§4), nAdroid uses the Soot [21] program analysis framework to generate a dummy main method. nAdroid uses the Android API listener-callback list from FlowDroid [1] to identify entry callbacks. For the potential ordering violation detection (§5), Chord [29], based on Datalog and the *bddbddb* solver [46], is used without MHP and lockset analyses. The sound and unsound filters (§6) are also implemented based on Chord. We ran nAdroid on a quad-core Intel i7 3.6 GHz system with 16GB of memory, running the 3.13 Linux kernel.

**Limitations** The current nAdroid implementation covers most common callbacks and threads, but does not yet support `Fragment`, `Layout`, and custom `Views`. In addition, the nAdroid prototype does not keep track of a specific looper thread on which each callback runs, and thus assumes that each component (e.g., `Activity`) has exactly one looper thread, and does not have other user-created looper threads. If a component has multiple looper threads, they break the atomicity assumption between callbacks, and the IG and IA filters should be downgraded to unsound filters. Among tested applications, the assumption holds for 17/27 applications. We have not confirmed whether the IG and IA filters unsoundly prune potential UAF bugs between multiple looper threads in the rest applications. Previous work [17, 39] shares this assumption of a single looper thread.

### 8.2 Tested Applications

We evaluated nAdroid with two groups of open-source Android applications, 27 in total, listed in Table 1. The first "train" group of 7 applications include all the applications that were used for evaluating CAFA [17], a dynamic UAF detector, except `Camera`, `VLC`, and `FBReader` where we could not locate their source code or analyze them with Soot [21]. The train group was used to learn common may-happens-before relation and programming idioms in Android and to design unsound filters (§6.2).

The second "test" group of 20 applications consist of 6 applications evaluated in another dynamic UAF detector DroidRacer [26], and 14 new applications randomly picked from [13] that collects the most popular Android applications. Three applications `Music`, `Browser`, and `Mytrack` were



**(a)** Sound filters. **(b)** Unsound filters (after sound).

**Figure 5.** Effectiveness of sound and unsound filters. Each filter is evaluated independently, so there is overlap.

evaluated in both CAFA and DroidRacer. Among them, `Mytrack` has significant version/source code change and all true UAF warnings found are different, thus is included in the test group with a different subscript number. Except Table 1, all the evaluation results are based on these 20 test applications.

In Table 1, the first three columns show the train/test group, the application name, and the size of the code base, sorted by LOC. The next EC and PC columns show the static number of Entry Callbacks and Posted Callbacks, respectively. The T column shows the static number of Threads in an application including the dummy UI main thread, `AsyncTask`'s `doInBackground` thread, and native Java threads.

### 8.3 Effectiveness of Sound and Unsound Filters

Table 1 summarizes the overall results of nAdroid's UAF analysis. The 7th column represents the total number of potential UAF warnings detected by static analysis (§5). Each warning is a pair of free-use operations. In this section, we first focus on the effectiveness of nAdroid filters.

The 8th and 9th columns of Table 1 show the number of remaining races after applying all the sound filters (§6.1) and unsound filters (§6.2) in sequence. On average, sound filters prune out 88% of potential UAF warnings; unsound filters suppress 70% of the remaining UAF warnings; and in combination these filters yield a 96% reduction.

Figure 5(a) shows the effectiveness of each sound filter when applied individually. Our novel MHB filter based on must-happen-before relation between event callbacks prunes out many false UAF warnings (21%). IG and IA filter are found to be effective: 66% and 13%, respectively. Note that there is some overlap between the sound filters. While the IG and IA filters are independent of each other, the MHB filter may overlap with the IG filter (5.9%) and the IA filter (6.7%) because MHB considers happens-before order at the event callback granularity.

Figure 5(b) shows the effectiveness of each unsound filter when applied individually to the UAF warnings that remain after applying the sound filters. Our novel mayHB filter based on may-happens-before relation effectively downgrades 13% of the remaining warnings, among which post-based PHB contributes the most (10%). The MA, UR, and TT filters prune

**Table 1.** The result of nAdroid's UAF analysis: filters, type of remaining UAFs, true harmful UAFs, and false UAF warnings.

| Type | APP | LOC | EC | PC | T | Potential UAFs Detected | Remaining UAFs after sound filters | Remaining UAFs after unsound filters | Type of Remaining UAFs | | | | | True harmful UAFs | False Positives | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | EC-EC | EC-PC | PC-PC | C-RT | C-NT | | Path insens. | Points-to | Not reach. | Missing HB | Not analyzed |
| Train | ToDoList | 2637 | 45 | 1 | 1 | 54 | 32 | 0 | | | | | | 0 | | | | | |
| | Zxing | 6453 | 65 | 15 | 14 | 263 | 6 | 2 | 2 | | | | | 0 | | | | 2 | |
| | Music | 10518 | 271 | 41 | 1 | 19167 | 2491 | 207 | 53 | 94 | 60 | | | 0 | 46 | | 14 | 2 | 145 |
| | MyTracks_1 | 27080 | 280 | 58 | 38 | 825 | 173 | 80 | | 1(1) | | 36(2) | 43(26) | 29 | 51 | | | | |
| | Browser | 30675 | 216 | 47 | 53 | 34185 | 8077 | 0 | | | | | | 0 | | | | | |
| | ConnectBot | 32645 | 105 | 31 | 19 | 197 | 33 | 13 | | 12(12) | 1(1) | | | 13 | | | | 4 | |
| | FireFox | 102658 | 748 | 28 | 135 | 16546 | 10004 | 1540 | | | | 1533 | 7(1) | 1 | 455 | 6 | | | 1078 |
| Test | SoundRecorder | 1194 | 14 | 0 | 1 | 9 | 0 | 0 | | | | | | 0 | | | | | |
| | Swiftnotes | 1571 | 32 | 1 | 1 | 0 | 0 | 0 | | | | | | 0 | | | | | |
| | PhotoAffix | 1924 | 52 | 9 | 2 | 84 | 10 | 4 | 2 | 2 | | | | 0 | 4 | | | | |
| | MLManager | 2073 | 153 | 11 | 10 | 304 | 38 | 0 | | | | | | 0 | | | | | |
| | InstaMaterial | 2248 | 42 | 29 | 4 | 6496 | 544 | 0 | | | | | | 0 | | | | | |
| | Tomdroid | 2372 | 24 | 4 | 3 | 0 | 0 | 0 | | | | | | 0 | | | | | |
| | SGT Puzzles | 2944 | 60 | 14 | 5 | 591 | 0 | 0 | | | | | | 0 | | | | | |
| | Aard | 3684 | 53 | 20 | 25 | 216 | 111 | 48 | | | | 8 | 40(8) | 8 | 9 | 15 | | 16 | |
| | ClipStack | 3948 | 106 | 18 | 2 | 4 | 0 | 0 | | | | | | 0 | | | | | |
| | KissLauncher | 5210 | 66 | 7 | 13 | 264 | 42 | 36 | 36 | | | | | 0 | 36 | | | | |
| | DashClock | 10147 | 67 | 13 | 1 | 74 | 1 | 0 | | | | | | 0 | | | | | |
| | Dns66 | 10423 | 22 | 4 | 6 | 99 | 13 | 13 | 1 | 12 | | | | 0 | 10 | | | 3 | |
| | CleanMaster | 11014 | 117 | 38 | 12 | 7 | 0 | 0 | | | | | | 0 | | | | | |
| | OmniNotes | 13720 | 764 | 19 | 22 | 10360 | 32 | 0 | | | | | | 0 | | | | | |
| | Solitaire | 15478 | 47 | 70 | 2 | 48 | 31 | 1 | 1 | | | | | 0 | 1 | | | | |
| | Mms | 27578 | 413 | 37 | 52 | 10439 | 3990 | 1207 | 258 | 139 | | 306 | 504 | 0 | 196 | 164 | | 3 | 844 |
| | MyTracks_2 | 37031 | 1029 | 59 | 52 | 1104 | 145 | 71 | | | | 10 | 61 (27) | 27 | 34 | 10 | | | |
| | MiMangaNu | 37827 | 24 | 9 | 10 | 10 | 1 | 0 | | | | | | 0 | | | | | |
| | QKSMS | 56082 | 225 | 37 | 35 | 536 | 171 | 19 | | 9 | | | 10(10) | 10 | 8 | | | 1 | |
| | K-9 Mail | 78437 | 499 | 27 | 20 | 45336 | 4143 | 918 | 34 | 148 | 24 | 418 | 294 | 0 | | 275 | | 2 | 641 |

26%, 29%, and 15% of the UAF warnings, respectively. Similar to MHB, we also found fine-grained filters MA and UR have overlapped coverage with the other coarse-grained filters, ranging from 1% to 9.6%. We present a false negative analysis on our unsound filters in §8.6.

## 8.4 Detecting True Harmful UAFs

From the remaining UAF warnings, we identified 88 harmful UAF bugs by manually constructing a dynamic execution leading to a `NullPointerException`. nAdroid provides the callback and thread sequence associated with each potential UAF bug (§7) so that programmers can reason about the root entry callbacks to start an exploration from. Based on these hints, we instrumented potential UAF warnings to perturb the schedule of event callbacks and threads. We used a timer for callbacks not to block the looper thread, and a spinloop for threads. Though we were able to find true UAFs successfully, we note that this manual validation process was indeed a time-consuming job. We expect that with nAdroid, the developers who are familiar with the code base can find harmful cases much easily as nAdroid provides many relevant information: e.g., racy variable, instruction, event, and call path, etc. Developing an automatic validation solution would be a good future work.

In Table 1, the 10th–15th columns show the "types of remaining UAFs" classified by the origins to which a pair of free/use operations belong, respectively; and the number in brackets represents the number of validated true harmful UAF violations in the category. Then, the next (16th) column shows the total number of "true harmful UAFs" that nAdroid found.

Among 88 harmful UAF bugs found, at least 58 were not previously reported in the literature. For the remaining 30 in `Mytrack_1` and `Firefox`, we could not confirm if they are novel because CAFA offers neither racy traces nor input to reproduce the results. CAFA reports no error between callbacks in `ConenctBot`. Previous dynamic tools [17, 26] miss them due to test coverage. Static tool DEvA [39] also has many false negatives because of limited analysis scope, which we discuss in §8.7.

Though we did not analyze all the cases (§8.5), our result empirically supports our hypotheses introduced in §7 as most true UAF races are found in cases where PC and NT are involved. The harmful UAF examples in Figure 1 represent all different cases: (a) is an EC-PC UAF in `ConnectBot`; (b) PC-PC in `ConnectBot`; and (c) C-NT in `FireFox`.

## 8.5 False Positive Analysis

Then, we inspect the rest cases to conduct false positive analysis. For the applications with relatively small number of remaining UAFs after filtering, we analyzed them all; whereas we sampled and analyzed 30% cases for `Music`, `Firefox`, `Mms`, and `K-9 Mail`.

In Table 1, the last five columns represent the four sources of false positives (and the number of warnings not analyzed):
**Path Insensitivity.** nAdroid analysis (including Chord) is path-insensitive as in other static analyses. We found it common to use high-level flag variable, and the program takes different path, making the manifestation of UAF warnings infeasible at runtime. This is the most common source.
**Points-to Analysis.** Like other static analyses, nAdroid also suffers from imprecision in points-to analysis. nAdroid takes advantage of Chord's k-objective sensitive points-to

analysis with the default value k=2 for balancing precision and scalability. We note that objects created by a static method (no context) does not take advantage of k-object-sensitive pointer analysis, becoming the common source of false positives.

**Not Reachable.** Either use or free operation of UAF warnings exists in a component that is not reachable by either explicit or implicit `intent`.

**Missing Happens-Before.** In Android, one event can enable/disable other events and it is especially common in UI callbacks. For example, one can set a button invisible in some specific condition. Statically capturing these UI interactions requires understanding semantics of a diverse set of Android APIs, and nAdroid misses such happens-before orders.

Note that the sources of false positives are not related to happens-before orders yet rather share the inherent limitations of static analyses. This highlights the effectiveness of nAdroid's (sound must-, unsound may-) happens-before-based filters. Another important implication is that the improvement of underlying static analyses such as Chord's points-to analysis makes it possible to increase the accuracy of nAdroid, though it is beyond the scope of this paper.

## 8.6 False Negative Analysis

This section investigates the sources of false negatives in nAdroid analysis. Ideally, false negative analysis requires ground truth to feed and test on. Unfortunately, there does not exist such a benchmark. As an alternative, we decided to inject artificial UAF violations and check if nAdroid detects them or not. Furthermore, instead of randomly adding UAFs, we took the true data races reported by DroidRacer[26], a trace-based dynamic data race detector, and introduced new UAF ordering violations at the same locations as original data races in order to construct more faithful and practical cases. In total, we were able to create 28 ground truth.

Table 2 shows the result of our false negative analysis with artificial UAFs on 8 test applications. The first column shows all the applications tested by DroidRacer. The next six columns show the diverse "types (based on origins) of artificially injected UAF violations", using the same classification as Table 1. The next column "All" presents the total number of UAF violations for each application. The last two columns presents UAFs that nAdroid could not detect.

nAdroid misses 5 (out of 28) UAFs for two reasons: unanalyzed code and unsound filter. First, nAdroid could not detect two UAFs in `Mms`. Further investigation reveals that `IBinder` object was passed to the Android framework and returned to the application later. Thus, nAdroid (based on Chord's call graph) was not able to track the full call graph and racy accesses happens along that call path. As a dynamic tool, DroidRacer was able to monitor a full interaction between application and the framework, which was not the

**Table 2.** False Negative Analysis

| APP | Type of artificial UAF Violations | | | | | All | Missed by detection | Pruned by unsound filters |
|---|---|---|---|---|---|---|---|---|
| | EC-EC | EC-PC | PC-PC | C-RT | C-NT | | | |
| Tomdroid | | | 1 | | | 1 | | |
| Puzzles | | 5 | | | 4 | 9 | | 1 |
| Aard | | | | | 1 | 1 | | |
| Music | | 2 | 4 | | | 6 | | |
| Mms | 2 | 3 | | | 1 | 6 | 2 | |
| Browser | 2 | | | | 1 | 3 | | 2 |
| MyTracks_2 | | 1 | | | | 1 | | |
| K-9 Mail | | | | 1 | | 1 | | |
| Total | 4 | 11 | 5 | 1 | 7 | 28 | 2 | 3 |

case for nAdroid. It is in theory possible for nAdroid to include the whole Android framework into the scope of static analysis, but it would suffer from scalability issue.

Second, nAdroid missed two UAFs in `Browser` and one UAF in `Puzzles`. It turns out that all three cases are due to the unsound CHB filter, especially filtered by `finish` method. The event callbacks used `finish` method to close the `Activity` as an error handling routine in a special program path. CHB filter relies on may analysis and unsoundly assumes a happens-before order if there exists at least one program path reaching `finish` method. Despite its unsoundness, our prior experimental results (§8.3 and §8.4) make the case for CHB filter. We do not find any other false negatives in other unsound filters. Lastly, in the next section, we report one more case that nAdroid was not able to detect due to implementation limitation.

## 8.7 Comparison to DEvA

This section compares nAdroid with the state-of-the-art static tool DEvA [39]. As described in §2.3, DEvA's unsound algorithm may lead to false negatives. In this section, we first investigate if nAdroid can find the UAF bugs that DEvA detects. Furthermore, we show that the lack of happens-before relation analysis in DEvA results in many false positives.

Table 3 includes the UAF warnings detected and marked as harmful by DEvA. The first column gives the name of applications, which are the same set as CAFA. The next four columns provide the details about each UAF warning: the racy field; the class; the name of callback including the use operation (e.g., `db.use()`); and the name of callback with the free operation (e.g., `db=null`). Then, the last two columns show if nAdroid finds the same UAF warning or not; and if it filters out the UAF warning or not.

When determining whether nAdroid detects the same UAF warning or not, we applied the sound IG and IA filters only (and not the other filters); then checked if the same UAF warning can be found in nAdroid's report. The reason is that DEvA classifies an UAF warning to be harmful if it is not protected by if-guard or intra-allocation, and thus the consistent definition is used.

We first show that nAdroid has fewer false negatives than DEvA. Table 3 shows that nAdroid can detect all UAFs that are marked as harmful by DEvA, except the last one in

**Table 3.** Comparison to DEvA

| APP | Field | Class | Use Callback | Free Callback | nAdroid |
|---|---|---|---|---|---|
| ToDoList | db | ToDoActv | onActvResult | done | Detected & Filtered |
| Music | mAdapter | AlbBrowActv | onActvResult | onDestroy | Detected & Filtered |
| | mAdapter | ArtAlbBrowActv | onRetNCfgIns | onDestroy | Detected & Filtered |
| | mPlayer | MediaPlayServ | setNextTrack | onDestroy | Detected & Filtered |
| | mAdapter | ArtAlbBrowActv | onActvResult | onDestroy | Detected & Filtered |
| | mAdapter | TrackBrowActv | onRetNCfgIns | onDestroy | Detected & Filtered |
| | mAdapter | AlbBrowActv | onRetNCfgIns | onDestroy | Detected & Filtered |
| | mAdapter | QueryBrowActv | onActvResult | onDestroy | Detected & Filtered |
| | mAdapter | TrackBrowActv | onActvResult | onDestroy | Detected & Filtered |
| | mAdapter | QueryBrowActv | onRetNCfgIns | onDestroy | Detected & Filtered |
| Mytracks_1 | binder | TrackRecServ | onBind | onDestroy | Detected & Filtered |
| | provUtils | TrackRecServ | onLocChgAsyc | onDestroy | Detected & Reported |
| Browser | mCtrlWV | AccessPrefFrag | onResume | onDestroy | Not detected |

`Browser`. As mentioned in Section 8.1, the current nAdroid prototype does not model `Fragment` yet, and thus the UAF warning in the class `AccesibilityPreferencesFragment` is not detected. In theory, this case can be detected with proper implementation. On the other hand, DEvA misses many UAF warnings detected by nAdroid. The applications (and their true positives) included in Table 1 (nAdroid's result), but not in Table 3 (DEvA's result) represent such cases. For instance, DEvA does not reports the harmful UAF bugs examples in Figure 1. (See §2.3 why DEvA suffers from false negatives.)

nAdroid also has fewer false positives than DEvA, thanks to its static happens-before analysis. The last column in Table 3 shows that nAdroid detected but filtered the 11 cases, and agrees only the one case as harmful races. Upon further investigation, we found that most (9/12) cases involving `onDestroy` are pruned by our sound MHB filter. The rest two cases are filtered by (unsound) CHB filter, and we manually validate that they are false warnings.

### 8.8 Analysis Execution Time

The execution time of nAdroid can be partitioned into 3 parts: modeling (§4), static detection (§5), and filtering (§6). On average, modeling took 32 seconds (1.19%) and filtering 82 seconds (3.08%). As expected, static detection took most of the static analysis time (about 42 minutes, 95.73%). The scalability of nAdroid would depend on Chord, which has shown to be able to handle programs with >180K LOC in the original paper [29]. If the execution time or scalability becomes an issue, then the k-value can be adjusted at the cost of precision.

## 9 Related Work

§2.3 described the most related race detectors for Android [3, 17, 24, 26, 39]. nAdroid can be applied to other concurrency bugs such as no-sleep bugs [33] and energy bugs [2] where racy API calls lead to ordering violations.

Static data race detection for traditional multi-threaded programs can be categorized into type system based [5, 14, 36], lockset analysis based [12, 29, 35, 43], or model based [37] approaches. Dynamic data race detectors [15, 40, 41, 48–50], runtime mitigations [25, 42, 47], testing techniques [16, 18] have been also proposed. However, as mentioned before, naive application of such solutions to Android programs would result in imprecise results due to lack of event-driven concurrency model in analyses.

As another domain, JavaScript/HTML web applications take single-threaded event-driven concurrency model. WebRacer [34] first defined the shared objects and the happens-before relationships specific to web applications. EventRacer [38] improved its scalability and introduced the concept of covered races to decrease the number of reported harmful races. Recently, Node.fz [10] extended concurrency analysis to Node.js, a server-side Javascript framework, that uses the single-threaded event loop and multithreaded worker pool.

## 10 Conclusion

This work presents nAdroid, a novel static ordering violation detector for Android applications. nAdroid models event callbacks as threads and detects ordering violations between callbacks and between threads in a unified manner. nAdroid also introduces novel happens-before-based filters to prune out false/benign warnings. Experimental results show that nAdroid detects harmful use-after-free ordering violations; and outperforms state-of-the-art technique.

## A Artifact Description

### A.1 Abstract

Our artifact provides all the runnable jar files of nAdroid and all the Android applications tested in this paper, along with scripts to use these to regenerate the results in our evaluation section. We tested our artifact on a quad-core Intel i7 3.6 GHz system with 16GB of memory, running the 3.13 Linux kernel. We also provide a virtual machine image with all dependencies installed. To validate the results, run the scripts and check the results according to the README file.

### A.2 Description

#### A.2.1 Checklist (Artifact Meta Information)

- **Program: java and datalog**
- **Compilation: ant and make**
- **Data set: android application package (apk)**
- **Output: a table described in a csv file**
- **Experiment workflow: install the dependencies, download and unpack the archive, run the scripts and observe the results or import the virtual machine image, run the scripts and observe the results**

#### A.2.2 How Delivered

The archive, virtual machine image and README file are available on:
Github https://github.com/VTLeeLab/CGO18-nAdroid-Artifact
Google Drive https://goo.gl/V12t34

### A.2.3 Software Dependencies

nAdroid requires Ant and Java with version 7. nAdroid has been tested on Ubuntu 14.04, and is expected to run correctly under other Linux distributions.

### A.2.4 Datasets

The artifact provides all the tested applications in the Evaluation section. It includes 35 apk files in total: 7 in the training set, 20 in test set, 8 in the artificial race injected set.

## A.3 Installation

If you want to run the artifact on your own computer, please install the following dependencies:

```
$ sudo apt-get install onpenjdk-7-jre
$ sudo apt-get install onpenjdk-7-jdk
$ sudo apt-get install ant
```

If you want to use the virtual machine image, please import the ova file into the Virtual Box.

## A.4 Experiment Workflow

For the convenience of the artifact evaluation, we provide a series of shell scripts which run the modeling, race detection, filtering and result analysis.

Below are the commands to test all the tested applications in the Evaluation section. It tests 35 applications in total: 7 in the training set, 20 in the test set, 8 in the the artificial race injected set.

```
$ cd nAdroid
$ ./run-all.sh
```

Running all the applications costs around 30 hours in a host desktop with an 4-core cpu. You can also use the following command to test the environment first. This command only tests 3 applications in total: 1 in the training set, 1 in the test set, 1 in the artificial race injected set.

```
$ ./run-all-test.sh
```

## A.5 Evaluation and Expected Result

After completing either of the above commands, a Result-Analysis.csv file is generated in the Result folder. It contains the data using in Figure 5 and Table 1 in the paper. The LOC information and manual inspection result in Table 1 are not provided in the ResultAnalysis.csv file.

The data of Table 2 exists in the Result/Injected folder. The data of Table 3 exists in the Result/Train folder. However, they all require manual inspection.

## Acknowledgements

The authors would like to thank the anonymous referees for their valuable comments, and James C. Davis for his careful proofreading. This work was in part supported by the National Science Foundation under the grant CCF-1527463, and the Google Faculty Research Award.

## References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 259–269.

[2] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 588–598.

[3] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 332–348.

[4] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. 2015. Droidel: A General Approach to Android Framework Modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2015)*. 19–25.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. 211–230.

[6] CVE-2008-0034. 2008. http://www.cvedetails.com/cve/CVE-2008-0034.

[7] CVE-2010-0923. 2010. http://www.cvedetails.com/cve/CVE-2010-0923.

[8] CVE-2010-1754. 2010. http://www.cvedetails.com/cve/CVE-2010-1754.

[9] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 145–160.

[10] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. 145–160.

[11] Evelyn Duesterwald and Mary Lou Soffa. 1991. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the symposium on Testing, analysis, and verification*. ACM, 36–48.

[12] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. 237–252.

[13] F-Droid. 2017. https://f-droid.org/.

[14] Cormac Flanagan and Stephen N. Freund. 2000. Type-based Race Detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. 219–232.

[15] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, Vol. 44. ACM, 121–133.

[16] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 174–186.

[17] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 326–336.

[18] Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 165–174.

[19] Jeff Huang and Arun K Rajagopalan. 2016. Precise and maximal race detection from incomplete traces. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 462–476.

[20] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 185–198.

[21] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[22] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[23] Lin Li and Clark Verbrugge. 2005. A Practical MHP Information Analysis for Concurrent Java Programs. In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing (LCPC'04)*. 194–208.

[24] Yu Lin, Cosmin Radoi, and Danny Dig. 2014. Retrofitting Concurrency for Android Applications Through Refactoring. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 341–352.

[25] Brandon Lucia and Luis Ceze. 2013. Cooperative empirical failure avoidance for multithreaded programs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 39–50.

[26] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 316–325.

[27] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. 1–11.

[28] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41.

[29] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. 308–319.

[30] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 22–31.

[31] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. 1999. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. 338–354.

[32] Best new octa-core Android smartphones (2015 edition). 2015. http://www.phonearena.com/news/Best-new-octa-core-Android-smartphones-2015-edition_id65222.

[33] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 2012. What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. 267–280.

[34] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 251–262.

[35] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. 320–331.

[36] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (Jan. 2011), 3:1–3:55 pages.

[37] Shaz Qadeer and Dinghao Wu. 2004. KISS: Keep It Simple and Sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. 14–24.

[38] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications (OOPSLA '13)*. 151–166.

[39] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. 2015. Detecting Event Anomalies in Event-based Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 25–37.

[40] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[41] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. 2011. RACEZ: a lightweight and non-invasive race detection tool for production applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 401–410.

[42] Kaushik Veeraraghavan, Peter M Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and surviving data races using complementary schedules. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. ACM, 369–384.

[43] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. 205–214.

[44] TJ Wala. 2014. Watson libraries for analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[45] Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, and Yong Guan. 2014. Localization of concurrency bugs using shared memory access pairs. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 611–622.

[46] John Whaley. 2007. *Context-sensitive Pointer Analysis Using Binary Decision Diagrams*. Ph.D. Dissertation. Stanford, CA, USA. Advisor(s) Lam, Monica. AAI3253554.

[47] Jingyue Wu, Heming Cui, and Junfeng Yang. 2010. Bypassing Races in Live Applications with Execution Filters.. In *OSDI*, Vol. 10. 1–13.

[48] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 221–234.

[49] Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical Data Race Detection for Production Use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 149–162.

[50] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2016. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 159–173.