

# Analysis of GUI and VUI Interaction Conflicts on Mobile Apps

**Abstract.** Mobile apps have infiltrated our daily lives. They largely rely on modern mobile operating systems (OSes) like Android to handle their user interfaces, which essentially implements a complex concurrency model. This may expose them to the high risk of races. The recent advancement of the automatic speech recognition (ASR) technology further exacerbates this risk, as apps largely start embedding the voice user interface (VUI). The VUI framework involves interaction of multiple threads, complicating the traditional UI scheme that is purely based on graphics, i.e., graphical user interface (GUI). Race conditions, named *GV-race* specifically in this context, may occur when GUI and VUI both access to the same resource simultaneously.

In this work, we present the first study of the GV-race problem on Android apps. We characterize the GV-race on its consequences, root causes and patterns. Different from the GUI which responds immediately, VUI naturally involves multiple threads, and takes seconds to process the input and respond. Therefore, the integration of GUI and VUI brings numerous and complex data races. Our characterization enables us to formally define GV-race. We then develop *Roma* (GV-race detector on mobile apps) to detect the GV-race automatically. *Roma* extracts the GUI and VUI related call graph through static program analysis, and generates a universal GV interaction graph using our pre-defined highly abstract primitives to represent GUI and VUI interactions. It introduces *happen-before* constraints to formally specify the *freeness of GV-race* with respect to the GV interaction graph, so that the detection of GV-race can be reduced to constraint solving with off-the-shelf SMT solvers. We apply *Roma* to analyze 810 apps that support both the GUI and the VUI. It finds 100% of apps that contain in-parallel write-write on GUI-VUI shared resources are subject to the GV-race, and proves that 78.0% of them to be true positive. Careless usage of six popular VUI SDKs can all lead to the GV-race.

## 1 Introduction

The development of artificial intelligence has propelled the enhancement of automatic speech recognition (ASR) technology. As a result, ASR technology has produced an evolution in Android apps. Traditional Android apps predominantly rely on the graphical user interface (GUI), where users interact by tapping the screen or entering text. The voice user interface (VUI), supported by the ASR technology, enables users to interact through voice, making it immediately welcomed by users. For example, 31% mobile users use voice search at least once a week as reported by Statista[1]. Many developers also have started incorporating VUI into their apps for accessibility[2,3].

Despite the accessibility provided by the integration of GUI and VUI, concerns on potential race conditions arise. As the VUI typically involves multi-threaded interactions to handle complex tasks such as speech collection and recognition, the waiting period is likely to experience GUI events. Actually, the VUI framework provided by popular SDKs makes it easy for races to happen. To illustrate a scenario of races during GUI and VUI interactions, consider the VUI framework of the Google VUI SDK[4]. The main thread posts an ASR task, which is run by the remote server. After obtaining the ASR results, the remote server sends them back to the main thread. As a result, the

VUI response returns after the ASR process. Under this VUI framework, the following situation may happen: the VUI action(e.g., say commands) happens before the GUI action(e.g., click a button), but the VUI response arrives after the GUI response. That is a typical scenario for GV-race.

Although numerous studies have investigated detecting event-driven races on GUI-based apps using dynamic[5,6,7] or static[8,9,10] analysis, concerns arising from the mixed presence of GUI and VUI have not been adequately studied. The large number of mobile users that use VUIs, along with the complexity of the VUI framework and GUI and VUI interactions, underscores the significance of investigating this emerging issue. The involvement of VUI introduces two new critical challenges to detect the GV-race.

**Challenge 1:** The multi-threaded and time-consuming nature of VUI increases the complexity of the concurrency model, affects the temporal relationship, and significantly increases the possibility of races. In addition, unlike in purely GUI-based apps where synchronization mechanisms are relatively mature, GUI-VUI developers may attempt diverse strategies to prevent the GV-race. However, to the best of our knowledge, there have been no related research accurately formalizing the GUI and VUI interactions. Due to the lack of the formalization, the GV-race has not been comprehensively defined, analyzed or prevented.

**Challenge 2:** The incorporation of VUI may exaggerate those challenges that exist in detecting techniques, such as the state space explosion. Previous race detection methods require comprehensive detection of GUI-based races on Android apps, so most of them capture all GUI events. Adding VUI events on top of this will make the original problem too complex to handle, leading to low scalability and low efficiency.

To address the above challenges, we seek to formalize, define and automatically detect the GV-race. For **Challenge 1**, we specifically define a set of primitives to describe GUI and VUI interactions, including primitives about threads, tasks, locks(including traditional locks and GV related locks), and GV interactions. Happen-before constraints between primitives are introduced to specify the freeness of GV-race. Based on primitives, we formally define and analyze the GV-race.

For **Challenge 2**, we propose *Roma* (GV-race detector on mobile apps), a lightweight pairwise GV-race detection framework. *Roma* contains three steps, i.e., extract the GUI and VUI related call graph, build the GV interaction graph, and check the GV interaction graph. We take a pair of GUI and VUI actions as a detection unit for the reduction of state space and analysis time. Pre-defined primitives are used to build the GV interaction graph for each pair of conflicting GUI and VUI actions. After adding happen-before constraints to the graph, our problem is reduced to checking if all constraints can be satisfied by the SMT solver[11].

Finally, we conduct studies to investigate *Roma*'s accuracy, efficiency, scalability and applicability. We then conduct a large-scale study with it to investigate the landscape and root causes of GV-race. *Roma* analyzes 810 apps that support both the GUI and the VUI, and discovers that 100% of GV co-write apps(contain in-parallel write-write on GUI-VUI shared resources) exhibit GV-race. Over 80% apps are analyzed in 10 minutes with a true positive rate of 78.0%.

**Contributions:** Contributions are summarized below.

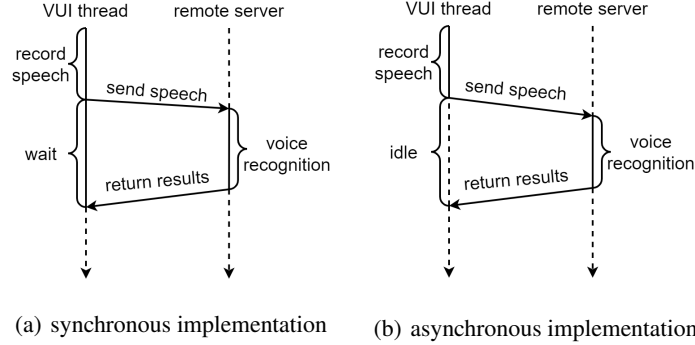


Fig. 1: VUI implementation patterns.

- **Analysis and definition of the GV-race.** We analyze the VUI framework of popular VUI SDKs and abstract their execution paradigms into two typical models. We also formally define the GV-race problem and analyze its root cause.
- **Definition of GV-race related primitives.** Except from traditional locks, developers may use strategies like disabling the GUI actions when the conflicting VUI actions are invoked to prevent the GV-race. Towards complex GUI and VUI interactions, we define a new set of primitives to cover all kinds of situations.
- **Pairwise GV-race detection framework.** We design Roma, a lightweight pairwise GV-race detection framework. It only considers races during GUI and VUI interactions and conducts the detection on the GUI and VUI related call graph. We take a pair of GUI and VUI actions as a detection unit to reduce the state space and analysis time. Roma analyzes 810 apps that support both the GUI and the VUI, and finds that 100% of GV co-write apps have the GV-race.

## 2 GV-race

We divide VUI implementations on Android apps into two categories(see section 2.1), and provide a motivating example(see section 2.2) to help understand the GV-race.

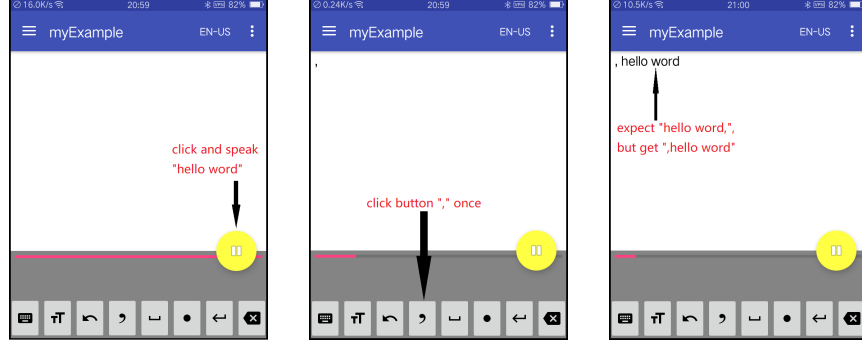
### 2.1 VUI Implementations on Android Apps

Typically, the VUI on Android apps is initiated when the user clicks a button and subsequently interacts with it through voice. The VUI records the user’s speech and transmits it to a remote server, which then converts it to text and sends the text back.

There are primarily two types of VUI implementations: synchronous and asynchronous. For the purpose of our discussion, we refer to the thread for initiating the VUI as the “VUI thread”. In the synchronous implementation(see figure 1(a)), the VUI thread is blocked until the remote server returns. Conversely, in the asynchronous implementation(see figure 1(b)), the VUI thread remains idle during the ASR process.

As part of our research, we investigated the VUI implementations of top-10 popular VUI SDKs according to Gartner’s study [12] in 2021, which rates the product or service for various language models. Six of them provide detailed implementations and sample apps. It is noteworthy that five SDKs(Google, Alibaba, Baidu, Tencent and Huawei) take the asynchronous strategy, while only Microsoft uses the synchronous implementation.

Despite the variety of VUI implementations, their commonalities involve multi-threaded interactions and long response duration. If the main thread(i.e., the thread to



(a) The user clicks the green button and speaks “hello word”. (b) The user clicks the “,” button, and the text box shows “,” immediately. (c) The ASR result returns, and the text box shows “,hello word”.

Fig. 2: A motivating example of the GV-race.

handle UI operations) is idle and conflicting GUI actions are not restricted during the ASR, the GV-race may arise.

## 2.2 A Motivating Example of the GV-race

In this section, we provide a specific example to demonstrate the GV-race. Figure 2 shows the “Voice Notebook - continuous speech to text” app, which uses the Google VUI SDK. To use it, the user clicks the button to start the VUI and says commands(see figure 2(a)). After a few seconds, the ASR result is appended to the text box. If the user presses the “,” button, a “,” will be appended to the text box(see figure 2(b)). However, if these two actions(i.e. saying commands and pressing the button) happen sequentially in a short time, “,” may display earlier than the ASR result unexpectedly(see figure 2(c)).

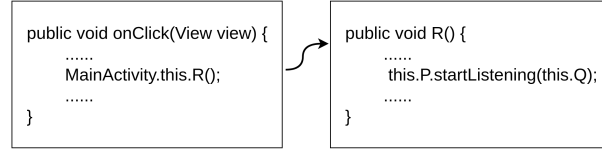
Next we explain how the race happens by referring to the source code. When the user clicks the button, the **onClick** function(Figure 3(a)) is executed, and it calls the **startListening** function in the Google VUI SDK to start the ASR. In the **startListening** function, the user’s speech is recorded and sent to the remote server. After that, the function returns and the main thread becomes idle. When the user presses the “,” button, the **btnCommaClick** function in Figure 3(b) is executed, which appends “,” to the text box. After that, the ASR result is returned by the callback function **onResults** (Figure 3(c)), which appends “hello world” to the text box. The GV-race happens as the developer does not restrict the conflicting GUI actions(clicking the “,” button) during the ASR process.

## 3 Formalization and Definition

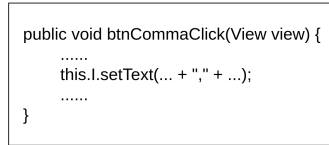
In this section, we define and analyze the GV-race by introducing functional level terms, primitives to abstract the GUI and VUI interactions and happen-before constraints between primitives.

### 3.1 Functional Level Terms

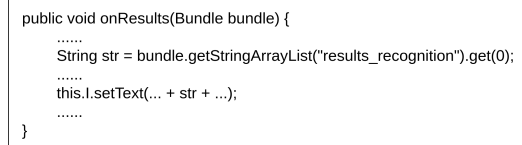
The GV-race arises on the premise of in-parallel write-write on GUI-VUI shared resources. So the functions and call relationships related to GUI/VUI actions are crucial



(a) The **onClick** method to start the VUI.



(b) The **btnCommaClick** method called after clicking the GUI button “,”.



(c) The **onResults** callback to process the ASR results.

Fig. 3: Source code of the motivating example.

for the GV-race. Below, we define terms of GUI and VUI related functions and call relationships at the source code level.

- **VUI source**: the function that processes the ASR results, e.g. **onResults**.
- **GUI source**: the function that is related to a GUI action, e.g. **btnCommaClick**.
- **Sink (VUI sink & GUI sink)**: the function that writes to the resource shared by GUI and VUI, e.g. **setText**.
- **VUI source call graph**: the call relationship of threads, tasks, locks or GV interactions related functions from the launch of VUI to the VUI source, e.g. **onClick** → **startListening** → **onResults**.
- **VUI sink call graph**: the call relationship of threads, tasks, locks or GV interactions related functions from the VUI source to the VUI sink, e.g. **onResults** → **setText**.
- **GUI sink call graph**: the call relationship of threads, tasks, locks or GV interactions related functions from the GUI source to the GUI sink, e.g. **btnCommaClick** → **setText**.
- **GUI and VUI related call graph**: the graph that represents call relationships given by the above mentioned call graphs.

With terms described above, we can demonstrate the GV-race at the source code level. In order to define such a race at a higher level, we introduce primitives to abstract the GUI and VUI interactions.

### 3.2 Primitives to Build GV Interaction Graphs

Race conditions occur when two operations access the same resource, and at least one of them is a write operation. Adding locks is a classic approach for preventing races. Other operations that may disable VUI/GUI actions, like starting a dialog, are referred to as GV-related locks in this paper. As a result, we design thread-related and lock-related primitives. Besides, asynchronous procedures are posted by tasks in the Android system, so we include task-related primitives. There are specific primitives for the GV-race. They are GV-related primitives.

To sum up, we define four types of primitives: thread-related, task-related, lock-related, and GV-related primitives. They are introduced below.

- **Thread-related primitives.** Thread-related primitives represent the operations to create, join, start, or end a thread. Five thread-related primitives are shown in Table 1.
- **Task-related primitives.** Task-related primitives represent the operations to post, start or end a task. Three primitives related to tasks are shown in Table 2.
- **Lock-related primitives.** Lock-related primitives represent disabling/enabling GUI/VUI actions, or locking/unlocking. Four lock-related primitives are shown in Table 3.
- **GV-related primitives.** GV-related primitives represent the ASR process, the VUI sink, and the GUI sink. Three GV-related primitives are shown in Table 4.

Thread-related	Meaning
<b><i>threadBegin</i></b> ( $t$ )	start the thread $t$
<b><i>threadEnd</i></b> ( $t$ )	end the thread $t$
<b><i>fork</i></b> ( $t, t'$ )	create the thread $t'$ by thread $t$
<b><i>join</i></b> ( $t', t$ )	the thread $t'$ joins to thread $t$
<b><i>executeTask</i></b> ( $t$ )	start executing tasks

Table 1: Thread-related primitives.

Task-related	Meaning
<b><i>postTask</i></b> ( $t, p, t'$ )	post a task $p$ to thread $t'$ by thread $t$
<b><i>taskBegin</i></b> ( $t, p$ )	start the task $p$ in the thread $t$
<b><i>taskEnd</i></b> ( $t, p$ )	end the task $p$ in the thread $t$

Table 2: Task-related primitives.

Lock-related	Meaning
<b><i>disable</i></b> ( $t, p$ )	disable executing the task $p$ in the thread $t$
<b><i>enable</i></b> ( $t, p$ )	enable executing the task $p$ in the thread $t$
<b><i>lock</i></b> ( $t, l$ )	acquire the lock $l$ in thread $t$
<b><i>unlock</i></b> ( $t, l$ )	release the lock $l$ in thread $t$

Table 3: Lock-related primitives.

GV-related	Meaning
<b><i>SpeechRecognition</i></b> ( $t$ )	thread $t$ transcribes the speech to text
<b><i>VUISink</i></b> ( $t$ )	the VUI sink in thread $t$
<b><i>GUISink</i></b> ( $t$ )	the GUI sink in thread $t$

Table 4: GV-related primitives.

These primitives are used to abstract GUI and VUI interactions and build the **GV interaction graph**. The GV interaction graph is the graph that represent behaviors of threads, tasks, locks and GV interactions at the primitive level. It is built by mapping functions in the GUI and VUI related call graph to primitives and adding happen-before constraints. Figure 4 shows the GV interaction graph of our motivating example. The corresponding GUI and VUI related call graph is placed on the left side for reference.

### 3.3 Happen-before Constraints

The happen-before relationships naturally exist between statements. These relationships always stand in any valid execution sequences, even under a race condition. Therefore, we define a series of happen-before constraints based on primitives. They are divided into three categories, namely sequence within tasks, sequence between tasks in the same queue, and sequence determined by semantics.

First, we introduce rules about the sequence within tasks. In the looper thread, primitives within each task are executed sequentially (see rule 1). In a non-looper thread, primitives are executed sequentially (see rule 2). We use  $a_i \leq a_j$  to represent that the primitive  $a_i$  happens before  $a_j$ . The function  $task(a_i)$  returns the thread and task of primitive  $a_i$ . The function  $thread(a_i)$  returns the thread of primitive  $a_i$ . The symbol “ $_$ ” represents any threads.

$$\frac{task(a_i) = task(a_j) = (t, p) \wedge i \leq j}{a_i \leq a_j} \quad (1)$$

$$\frac{\text{executeTask}(\_) \notin \{a_1, \dots, a_n\} \wedge \text{thread}(a_i) = \text{thread}(a_j) = t}{a_i \leq a_j} \quad (2)$$

Second, we introduce rules about the sequence between tasks in the same queue. Tasks in the same queue are executed in a first-in, first-out order. Rule 3 means that if task  $p_1$  is posted to the same thread before task  $p_2$ , the end of  $p_1$  happens before the start of  $p_2$ . Rule 4 means that if we know that a primitive in task  $p_1$  happens before a primitive in task  $p_2$  in the same thread, we can predict that the end of  $p_1$  happens before the start of  $p_2$ .

$$\frac{\text{postTask}(\_, p_1, t) \leq \text{postTask}(\_, p_2, t)}{\text{taskEnd}(t, p_1) \leq \text{taskBegin}(t, p_2)} \quad (3)$$

$$\frac{\text{task}(a_m) = (t, p_1) \wedge \text{task}(a_n) = (t, p_2) \wedge a_m \leq a_n}{\text{taskEnd}(t, p_1) \leq \text{taskBegin}(t, p_2)} \quad (4)$$

Finally, we define rules about the sequence determined by semantics. Rule 5 means the post of task  $p$  to thread  $t$  must happen before the begin of task  $p$ . Rule 6 means the creation of thread  $t'$  must happen before the begin of thread  $t'$ . Rule 7 means the end of thread  $t'$  must happen before its joint to thread  $t$ . Rule 8 means that the post of a task  $p$  either happens before the operation to disable it, or happens after the enable operation corresponding to the disable operation. Rule 9 means that either thread  $t$  releases the lock before thread  $t'$  acquires it, or thread  $t'$  releases the lock before thread  $t$  acquires it. Rule 10 represents the transitivity feature of the happen-before relationship.

$$\text{postTask}(t, p, t') \leq \text{taskBegin}(t', p) \quad (5)$$

$$\text{fork}(t, t') \leq \text{threadBegin}(t') \quad (6)$$

$$\text{taskEnd}(t', t) \leq \text{join}(t', t) \quad (7)$$

$$\text{enable}(t, p) \leq \text{postTask}(\_, p, t) \vee \text{postTask}(\_, p, t) \leq \text{disable}(t, p) \quad (8)$$

$$\text{unlock}(t, l) \leq \text{lock}(t', l) \vee \text{unlock}(t', l) \leq \text{lock}(t, l) \quad (9)$$

$$\frac{a_i \leq a_k \wedge a_k \leq a_j}{a_i \leq a_j} \quad (10)$$

In figure 4, we add part of the happen-before constraints to the GV interaction graph.

### 3.4 GV-race Definition

Given the primitives in section 3.2 and the happen-before constraints in section 3.3, we define the GV-race in definition 1.

**Definition 1** Given a GUI task from  $\text{postTask}(\text{main}, \text{click\_GUI})$  to  $\text{GUIsink}(\_)$ , a VUI task from  $\text{postTask}(\text{main}, \text{click\_VUI})$  to  $\text{VUIsink}(\_)$ ,  $\text{GUIsink}(\_)$  and  $\text{VUIsink}(\_)$  access the same object, and the constraint  $\text{postTask}(\text{main}, \text{click\_VUI}) < \text{GUIsink}(\_) \wedge \text{postTask}(\text{main}, \text{click\_GUI}) < \text{VUIsink}(\_)$  is satisfied, the **GV-race** happens.

According to our research, conflicting GUI and VUI responses are mostly write-write operations. In addition, a VUI task is more time-consuming than a GUI task. Without loss of generality, we focus on the VUI-before-GUI write-write GV-race.

**Definition 2** Given a GUI task from  $\text{postTask}(\text{main}, \text{click\_GUI})$  to  $\text{GUIsink}(\_)$ , a VUI task from  $\text{postTask}(\text{main}, \text{click\_VUI})$  to  $\text{VUIsink}(\_)$ , the  $\text{GUIsink}(\_)$  and  $\text{VUIsink}(\_)$  access the same object, and the constraint  $\text{postTask}(\text{main}, \text{click\_VUI}) < \text{postTask}(\text{main}, \text{click\_GUI}) \wedge \text{GUIsink}(\_) < \text{VUIsink}(\_)$  is satisfied, the **VUI-before-GUI write-write GV-race** happens.

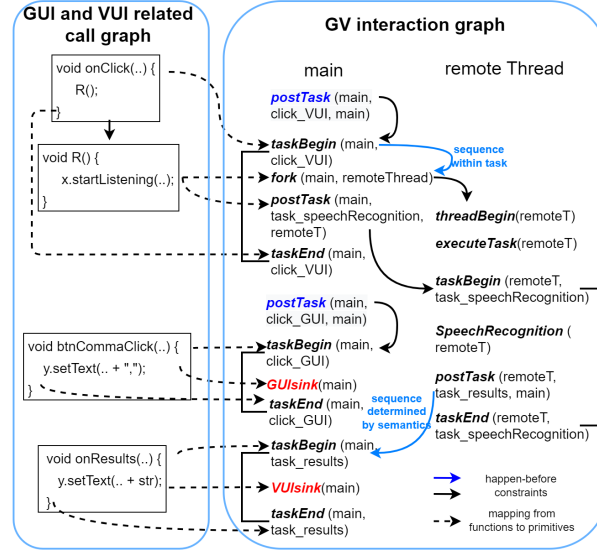


Fig. 4: Build the GV interaction graph from the GUI and VUI related call graph.

If the constraint in the Definition 2 is satisfied on the GV interaction graph, the GV-race happens. For example, we can add the GV-race constraint to the GV interaction graph in figure 4 and check. The GV-race constraint is satisfied on this GV interaction graph if the primitives are executed in top-to-bottom order.

## 4 Pairwise GV-race Detection

In this section, we introduce Roma, a lightweight pairwise GV-race detection framework.

### 4.1 Pairwise Detection Framework

Previous race detection techniques on Android apps mostly adopt an integrated modelling approach to detect GUI-based races. Adding VUI events on top of this makes the original problem too complex to handle. To efficiently detect the GV-race, we only retain GUI events that may interact with the VUI. We take a pair of GUI and VUI actions as a detection unit to further reduce the state space and analysis time.

Based on these ideas, we propose a lightweight pairwise GV-race detection framework called Roma. Figure 5 shows the overall framework of Roma. Roma starts by extracting the GUI and VUI related call graph. Based on this sub-call graph, Roma builds the GV interaction graph for each pair of conflicting GUI and VUI actions. Finally, the SMT solver [11] is used to check the GV interaction graph. We divide the above analysis process into three parts, extract the GUI and VUI related call graph, build the GV interaction graph, and check the GV interaction graph.

**Step1: Extract the GUI and VUI Related Call Graph.** Given an apk file, Roma takes the SPARK algorithm provided by FlowDroid [13] to build the entire call graph. Through static analysis, Roma extracts the GUI and VUI related call graph. This process will be detailed in Section 4.2.



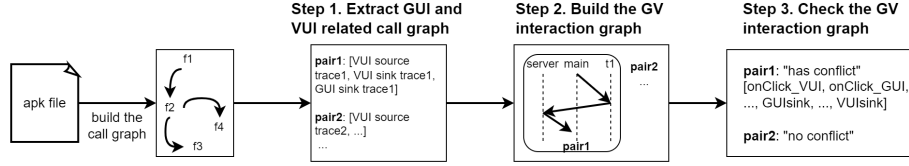


Fig. 5: The overall framework of Roma.

**Step2: Build the GV Interaction Graph.** Step2 takes the output of the step1 as the input. Roma maps functions to primitives and adds happen-before constraints to build a GV interaction graph for each pair of GUI and VUI actions that access the same resource. This process will be detailed in section 4.3.

**Step3: Check the GV Interaction Graph.** Step3 takes the output of the step2 as the input. Roma adds the GV-race constraint in Definition 2 to the GV interaction graph. Finally, we use the SMT solver [11] to check if all constraints can be satisfied. The result is recorded to avoid double checking the same GV interaction graph.

#### 4.2 Extract the GUI and VUI Related Call Graph

By exploring the call graph generated by FlowDroid [13], we extract the GUI and VUI related call graph. Figure 6 shows this process.

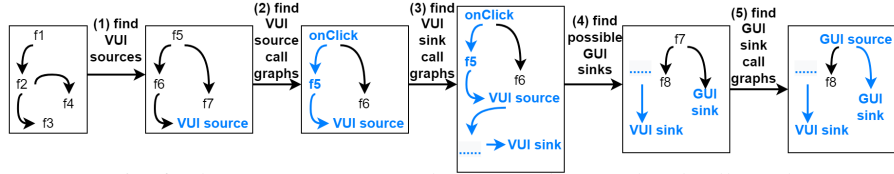


Fig. 6: The process to extract the GUI and VUI related call graph.

1. **find VUI sources.** Roma locates the VUI sources in the call graph, such as **onResults**.
2. **find VUI source call graphs.** Starting from a VUI source, Roma traverses through the call graph backward until a callback related to the user's action is found. Call relationships from the callback to the VUI source form the VUI source call graph.
3. **find VUI sink call graphs.** Starting from a VUI source, Roma traverses through the call graph forward until all functions related to write operations are found. These functions are recorded as VUI sinks. Call relationships from the VUI source to the VUI sink form the VUI sink call graph.
4. **find possible GUI sinks.** Roma locates the object written by the VUI sink in (3). Then, it uses alias analysis to identify statements that write to the same object, and records these statements as GUI sinks.
5. **find GUI sink call graphs.** Starting from a GUI sink, Roma traverses through the call graph backward to find a callback related to user's actions. Such a callback is recorded as the GUI source. This path is recorded as the GUI sink call graph.

Through the above five steps, we can locate pairs of GUI and VUI actions that access the same resource and their corresponding VUI source call graph, VUI sink call graph and GUI sink call graph. These call graphs form the GUI and VUI related call graph.

#### 4.3 Build the GV Interaction Graph

In step2, Roma builds the GV interaction graph for each pair of GUI and VUI actions that access the same resource based on the GUI and VUI related call graph. The GV

Function / sign	primitives
at the start of thread main	<b>threadBegin</b>
if a new thread $t'$ is forked by <b>fork</b> ( $\_, t'$ ), add to the start of thread $t'$	
at the end of a thread	<b>threadEnd</b>
<java.lang.Thread: void start()>	<b>fork</b>
<android.os.AsyncTask: void execute(...)>	
<java.lang.Thread: void join()>	<b>join</b>
<android.os.Looper: void loop()>	<b>executeTask</b>
<android.os.Handler: boolean post(...)>	<b>postTask</b>
<android.os.Handler: boolean postDelayed(...)>	
<android.app.Activity: void runOnUiThread(...)>	
when user-behavior-related callbacks are invoked, e.g. onClick	
if a new task $p$ is posted by <b>postTask</b> ( $\_, p, \_)$ , add to the start of task $p$	<b>taskBegin</b>
at the end of a task	<b>taskEnd</b>
start a new dialog / activity that disables the operation on the widgets on the previous interface, e.g. <android.app.Dialog: void show()>	<b>disable</b>
close a new dialog / activity that enables the operation on the widgets on the previous interface, e.g. <android.app.Dialog: void dismiss()>	<b>enable</b>
<java.util.concurrent.locks.ReentrantLock: void lock()>	<b>lock</b>
<java.util.concurrent.locks.ReentrantReadWriteLock\$WriteLock: void lock()>	
at the start of the code surrounded by synchronized	
<java.util.concurrent.locks.ReentrantLock: void unlock()>	<b>unlock</b>
<java.util.concurrent.locks.ReentrantReadWriteLock\$WriteLock: void unlock()>	
at the end of the code surrounded by synchronized	
used in VUI frameworks	<b>Speech-Recognition</b>
<b>VUI sink</b> statement, e.g. <b>setText</b>	<b>VUIsink</b>
<b>GUI sink</b> statement, e.g. <b>setText</b>	<b>GUIsink</b>

Table 5: From functions to primitives.

interaction graph is built by mapping the functions to primitives and inserting the VUI framework into the corresponding location. Most VUI SDKs are not open-source for developers. To address this issue, we decompile official apk files that use the same VUI SDKs and manually model the VUI implementation. When building the GV interaction graph, Roma embeds the pre-built VUI framework directly. Table 5 show sthe approach to mapping the functions to primitives defined in section 3.2.

Given the rules defined in section 3.3, we can add the constraints to the GV interaction graph. We use a variable to represent each primitive. If we predict that primitive  $a_i$  happens before primitive  $a_j$ , we add the constraint  $a_i < a_j$ .  $a_i < a_j$  implies that  $a_i$  happens before  $a_j$ . Algorithm 1 shows how we generate the GV interaction graph from call relationships in the GUI and VUI related call graph. It work as follows:

1. First, we set the main thread id  $mainT$  to “0”, the maximum thread id  $maxT$  to 1, the maximum task id to 0,  $graph$  to empty (line 1), and the constraint  $C$  as True (line 2).
2. From every entry function  $funct$  in  $crs$ , we start to add primitives by **PROCESSFUNC** function (lines 3-5).
3. In **PROCESSFUNC**, we find the primitive  $prim$  of function  $funct$  (**PROCESSFUNC**, line 3). If the  $prim$  is **fork**, we get a new thread id, add **fork** and **threadBegin** to  $graph$ . Then, we find the body of the new thread, and call **PROCESSFUNC**. After that, we

add **threadEnd** to *graph* (PROCESSFUNC, line 4-11). If the *prim* is **postTask**, the procedure is similar (PROCESSFUNC, lines 12-20). For other primitives, we just add the *prim* to the *graph* (line 22-23). If the function starts the VUI, the pre-built VUI framework is inserted to *graph* (line 24-26). Finally, we call PROCESSFUNC to process the rest call graph (PROCESSFUNC, lines 27-29).

4. After PROCESSFUNC returns, we get all the primitives. Then, we add constraints according to rules (lines 7-14). For each rule, we get its requirement *require* and result *result* (lines 8-9). If the *require* is satisfied, we add the *result* to the constraint *C* (lines 10-11). Otherwise, we add the constraint  $\neg \text{require} \vee \text{result}$  to *C* (lines 12-13).
5. Finally, the GV interaction graph *graph* and the happen-before constraints *C* are returned (line 16).

---

**Algorithm 1** Build the GV interaction graph based on the GUI and VUI related call graph

---

**Input:** Call relationships in the GUI and VUI related call graph *crs*, mappings from functions to primitives *map*, the list of rules *rules*

**Output:** The GUI interaction graph: *graph*, and the happen-before constraint *C*.

```

1: mainT  $\leftarrow$  "0"; maxT  $\leftarrow$  1; maxTask  $\leftarrow$  0; graph  $\leftarrow$  {}
2: C  $\leftarrow$  True                                     ▶ initialize the constraints.
3: for funct in crs.entry() do
4:   curT  $\leftarrow$  mainT; curTask  $\leftarrow$  GETNEWID(maxTask)           ▶ get a new task id
5:   PROCESSFUNC(funct, curT, curTask, crs)                 ▶ add primitives to the graph
6: end for
7: for rule in rules do                                     ▶ add constraints according to rules
8:   require  $\leftarrow$  rule.getRequire()
9:   result  $\leftarrow$  rule.getResult()
10:  if graph.meetReq(require) then
11:    C  $\leftarrow$  C  $\wedge$  result
12:  else
13:    C  $\leftarrow$  C  $\wedge$  ( $\neg \text{require} \vee \text{result}$ )
14:  end if
15: end for
16: return graph, C

```

---

```

1: function PROCESSFUNC(funct, curT, curTask, crs)
2:   newT  $\leftarrow$  curT; newTask  $\leftarrow$  curTask
3:   prim  $\leftarrow$  map.getPrim(funct); primName  $\leftarrow$  prim.getName()
4:   if primName == "fork" then                               ▶ fork a new thread and start it
5:     newT, maxT  $\leftarrow$  GETNEWID(maxT)                       ▶ get a new thread id
6:     graph.addFork(curT, newT)
7:     graph.addThreadBegin(newT)
8:     for newF in crs.funcOutOf(funct) do
9:       PROCESSFUNC(newF, newT, curTask, crs)           ▶ process funcs of the new thread
10:    end for
11:    graph.addThreadEnd(newT)
12:  else if primName == "postTask" then                       ▶ post a new task and start it
13:    newTask, maxTask  $\leftarrow$  GETNEWID(maxTask)             ▶ get a new task id
14:    postT  $\leftarrow$  funct.getPostThread()
15:    graph.addPostTask(curT, newTask, postT)
16:    graph.addTaskBegin(postT, newTask)

```

---

```

17:     for newF in crs.funcOutOf(funct) do
18:         PROCESSFUNC(newF, postT, newTask, crs)    ▶ process funcs in the new task
19:     end for
20:     graph.addTaskEnd(postT, newTask)
21: else
22:     if primName != "" then
23:         graph.addPrim(primName, curT)
24:     else if isStartVUIFunc(funct) then
25:         graph.addVUIFramework()
26:     end if
27:     for newF in crs.funcOutOf(funct) do
28:         PROCESSFUNC(newF, curT, curTask, crs)    ▶ process next funcs
29:     end for
30: end if
31: end function

```

---

#### 4.4 Check the GV Interaction Graph

Based on the GV interaction graph with constraints, the detection of GV-race can be reduced to solving constraints with SMT solvers. Algorithm 2 outlines how we detect the GV-race based on the GV interaction graph. It works as follows:

1. We initialize the result *result* to be “no conflict” (line 1). Each variable in *V* represents a primitive in the *graph* (lines 2).
2. We add the GV-race constraint *GV\_Constraint* to *C* (line 3). The solver *Solver* contains the set of variables *V* and the constraint *C* (line 4).
3. Finally, we can use the SMT solver to check whether all the constraints can be met (line 5). If they can, we set the *result* to “has conflict”, and the counterexample *counterExample* to the list of variables sorted by their values (lines 6-9). The results are returned (line 10).

---

**Algorithm 2** Detect the GV-race based on the GV interaction graph

---

**Input:** The GV interaction graph *graph*, happen-before constraints *C*, the GV-race constraint *GV\_Constraint*

**Output:** The result: *result*, the counterexample trace *counterExample*.

```

1: result ← “no conflict”; counterExample ← NULL
2: V ← graph.getPrims()    ▶ initialize the variables.
3: C ← C ∪ GV_Constraint
4: Solver ← {V, C}    ▶ variables, constraints
5: Solver.check()    ▶ SMT solver checks the constraints
6: if Solver.isSatisfied() then
7:     relationship ← “has conflict”
8:     counterExample ← V.sort()
9: end if
10: return result, counterExample

```

---

## 5 Evaluation

We implement Roma to detect the GV-race on Android apps and evaluate its performance in terms of accuracy, efficiency, scalability, and applicability. We also analyze the landscape and root cause of the GV-race.

## 5.1 Settings

**Dataset:** We scan the top-500 downloads in each category from apkpure [14]. Excluding factors such as network errors, we obtain over 10,000 apps and find 810 of them contain VUIs that are implemented by using Google Android SDK. Therefore, our data set is composed of the apk files of these 810 apps. We evaluate Roma on this data set.

The evaluation was run on the Ubuntu 20.04.3 machine with Intel® Core™ i7-10700 Processor CPU@2.9GHz and 16GB RAM. We make the source code and complete experimental results online, to facilitate future research on GV-race [15].

## 5.2 The Evaluation of Roma

Our evaluation aims to answer following research questions.

**RQ1: Landscape.** What is the landscape of the GV-race on apps that support both the GUI and the VUI?

**RQ2: Accuracy.** How is the accuracy performance of Roma?

**RQ3: Efficiency.** How long does Roma spend analyzing an app?

**RQ4: Scalability.** How is the size of apps analyzed in 10min, 30min and over 30min?

**RQ5: Root cause.** What is the root cause of GV-race?

**RQ6: Applicability.** Is Roma applicable to apps implemented by other VUI SDKs?

**Study 1: Landscape** We run Roma to analyze 810 apps. The time limit to analyze each app is set as 30 minutes. Out of 810 apps, 649 apps terminate successfully, while 161 apps failed. Among the 161 apps without results, 17 apps reported errors during the call graph construction, and 144 apps required longer analysis time.

total	terminate	GV co-write	conflict	conflict / GV co-write	conflict / terminate
810	649	59	59	100%	9.1%

Table 6: Number of apps with GV-race. GV co-write: contain in-parallel write-write on GUI-VUI shared resources

Table 6 shows the number of apps with GV-race detected by Roma. 100%(59/59) of apps that GV co-write have the GV-race. 9.1%(59/649) of apps that contain both the GUI and the VUI have the GV-race. The results show that the GV-race exists broadly in the apps that GV co-write.

**Answer to RQ1:** 100% of apps that GV co-write are detected to have the GV-race. The GV-race exists broadly in the apps with complex GUI and VUI interactions.

**Study 2: Accuracy** We manually verify the results of 59 GV co-write apps(see table 6) to evaluate Roma’s accuracy. Roma finds that all 59 apps have the GV-race. We prove 46 of 59 apps with GV-race (78.0%) through manual configuration. Due to space limitation, the case studies of real-world apps with GV-race are shown on our website[15].

The reasons for the false positives come from two parts. First, due to the existence of variable and function polymorphisms, it is difficult to accurately obtain the specific callees of all function calls. Another reason is that functions can be unreachable even they are in the call graph. However, solving these problems are both complex and time-consuming but cannot improve the accuracy evidently.

**Answer to RQ2:** The accuracy of Roma is high, with a true positive rate of 78.0% on GV co-write apps.

**Study 3: Efficiency** We record the time to analyze each app to measure the efficiency. Figure 7 shows the distribution of the number of apps over the analysis time. We can find that the number of apps shows an evident downward trend with the increase of analysis time. Most (about 80.7%) apps are analyzed in 10 minutes. The time-consuming task mainly lies in the construction of the call graph, which is done by FlowDroid.

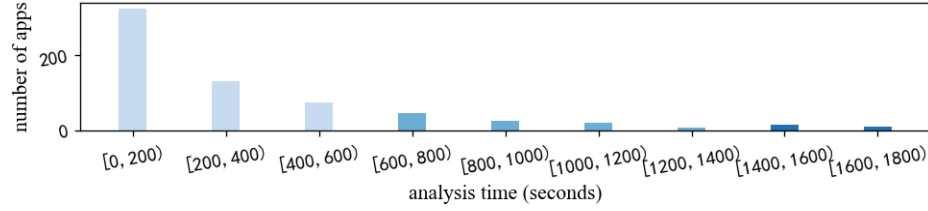
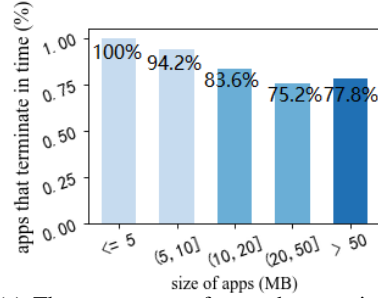


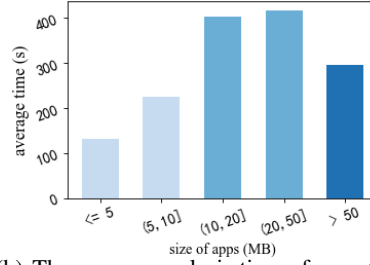
Fig. 7: The efficiency of Roma.

**Answer to RQ3:** Roma analyzes most apps efficiently. Over 80% of apps can terminate in 10 minutes.

**Study 4: Scalability** We study the scalability of Roma by comparing the analysis time of apps of different sizes. Figure 8(a) and 8(b) show the on-time completion rate and average analysis time for apps of different sizes.



(a) The percentage of apps that terminate on time for apps of different sizes.



(b) The average analysis time of apps that terminate on time for apps of different sizes.

Fig. 8: On-time completion rates and average analysis time for apps of different sizes.

As the size of apps increases, the on-time completion rate slowly decreases while the average analysis time slowly increases. The on-time complete rate remains at a level higher than 75%. The highest average analysis time is around 400 seconds.

**Answer to RQ4:** The on-time complete rate is higher than 75% for all sizes of apps.

**Study 5: Root cause** We analyze the features of the apps with GV-race to find the root cause. Their GV interaction graphs are divided into two patterns, one does most tasks in the main thread, and the other involves more threads/tasks. Figure 9(a) and 9(b) show the sketches of GV interaction graphs of these two patterns. The number of apps with pattern 1 and pattern 2 is 45 and 1 respectively.

In pattern 1, the main thread is used to start the ASR task. In the synchronous implementation, the VUI thread waits until the ASR results return, making it impossible to insert a GUI action. Although the GV-race may not happen under pattern 1 using the synchronous VUI implementation, the blocking of the main thread affects users'

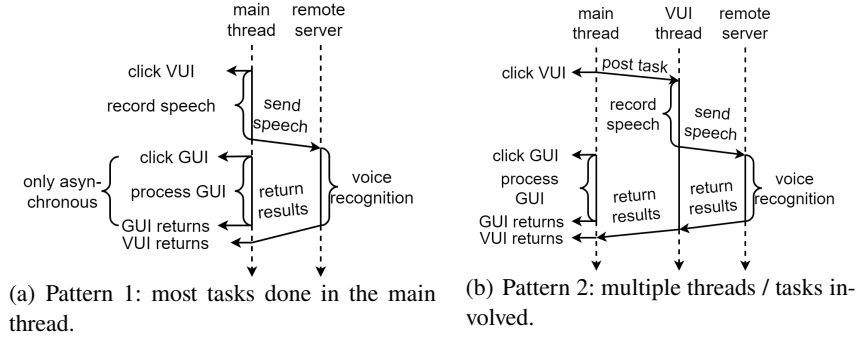


Fig. 9: Patterns of apps with GV-race.

experience. In pattern 2, the main thread starts another thread for the ASR. The main thread is idle during the ASR under both VUI implementations. Therefore, the GV-race will always happen under pattern 2 without proper strategies.

**Answer to RQ5:** The root cause of GV-race is that the main thread is idle while conflicting GUI actions are not restricted during the ASR.

**Study 6: Applicability** To show the applicability of Roma on other VUI apps, we also analyze apps that are implemented by using other VUI SDKs mentioned in section 2.1. We download the official sample apps given by these VUI SDKs’ developers and mutate them based on patterns in study 4 to form a data set. The mutated apps can be found on our website [15].

Roma finds that 9 apps have the GV-race of 10 mutated apps. The only app without the GV-race is the mutated Microsoft sample app under pattern 1. The Microsoft VUI SDK uses synchronous implementation, so the main thread is blocked when the remote server runs the ASR task. Our manual verification proves that the result is 100% true. This study shows that other VUI SDKs can also lead to the GV-race without careful design and Roma is perfectly suited for analyzing apps implemented by other VUI SDKs.

**Answer to RQ6:** VUI SDKs given by Alibaba, Microsoft, Baidu, Tencent, and Huawei can all cause the GV-race without careful design. Since Roma can detect 100% of the mutated sample apps’ GV-race, it is applicable to other VUI apps.

## 6 Discussions

### 6.1 Insights on the prevention of GV-race

We discover that developers are not broadly aware of the prevention of GV-race. In addition, we find that careless use of six popular VUI SDKs can all lead to GV-races.

We find 43 apps that opens a dialog after the VUI is invoked and closes it after the result returns out of 59 GV co-write apps. Among them, 40 apps use the variant version of the Google VUI SDK. However, GV-race may still happen if the attacker uses a script to invoke conflicting VUI and GUI actions in sequence quickly. Developers are encouraged to add traditional locks to prevent the GV-race.

### 6.2 Limitations and Threats to Validity

The limitations of Roma come from three parts. Firstly, functions in the call graph are not always reachable. Besides, Roma cannot detect GV-race on VUI apps implemented by

other VUI SDKs, since we model the VUI framework manually. Finally, the read-write GV-race are not included, but they can be detected with the similar approach.

## 7 Related Work

**Race detection on mobile apps.** Hsiao et al. [5] presented CAFA to check races between high-level operations in the event-driven mobile systems. DroidRacer [6] is a dynamic race detection technique that formalizes the concurrent semantics of the Android programming model. AATT+ [7] that extended AATT [16] works by exploring the app to find potentially conflicting resource accesses, and then pressure-testing them to detect concurrency bugs. More works [17,18,19,20] conducted automated GUI testing by exploring the app to cover more code and detect problems. All these approaches are based on dynamic analysis and may lead to high false negatives.

Some researches use static analysis to detect event-driven races. DeVA [8] detects the “event anomalies” where read-write or write-write operations access the same resources. SIERRA [9] automatically generates order among lifecycle and GUI events using a harness-based model, and used symbolic analysis to order the actions and memory accesses. ER Catcher [10] is a flow-, context- and thread-sensitive static analysis framework. It proposes the concurrency aware summary function to model the Android framework and leverages the Vector Clock method to detect happens-before relations.

Compared with Roma, the above techniques fail to model the VUI at the SDK level. Since they do not provide any primitives to describe actions like disabling/enabling the VUI, they can not accurately model the VUI and GUI interactions.

**Race conditions in traditional areas.** Several researches defined [21] and classified [22] race conditions. Farah et al. [23] studied the impact of data races on UNIX systems. Researches [24,25] detected the data race when two processes access the same file in the Unix environment. Another research [26] managed to detect and prevent the data race in file systems. Flanagan et al. [27] presented rccjava to detect race conditions in small to medium-scaled java programs and later extended it for large, realistic programs [28]. RacerX [29] uses flow-sensitive, interprocedural analysis to detect race conditions in large, complex multi-threaded systems. Licker et al. [30] proposed to use the build fuzzing method to detect missing dependencies when building the project. If the dependency between an input and an output is unknown, the job generating the input may conflict with the job reading it. WebRacer [31] and EventRacer [32] adapt the happen-before analysis to web and event-based applications to detect races.

## 8 Conclusion

In this paper, we propose, formalize and define the GV-race on Android apps. The GV-race happens when the VUI action happens before the GUI action, but the VUI response arrives after the GUI response. To detect the GV-race, we propose the Roma framework. Roma explores the call graph to extract the GUI and VUI related call graph. Then, it builds the GV interaction graph using pre-defined primitives and happen-before constraints. Finally, it checks if GV-race can happen on the GV interaction graph. Our experiments show that Roma achieves an accuracy rate of 78.0%. We use Roma to analyze 810 Android apps with both GUI and VUI and find 100% of apps that GV co-write with GV-race. Our work reveals that careless usage of VUI SDKs can lead to the GV-race and developers are not aware of such a problem.



## References

1. Frequency with which smartphone owners use voice-enabled technology worldwide in 2017. <https://www.statista.com/statistics/787382/worldwide-voice-technology-utilization/>, 2017.
2. Mobile accessibility at w3c. <https://www.w3.org/WAI/standards-guidelines/mobile/>, 2018.
3. José-Manuel Díaz-Bossini and Lourdes Moreno. Accessibility to mobile interfaces for older people. In Manuel Pérez Cota, João Barroso, Simone Bacellar Leal Ferreira, Benjamim Fonseca, Tassos A. Mikropoulos, and Hugo Paredes, editors, *Proceedings of the 5th International Conference on Software Development for Enhancing Accessibility and Fighting Info-exclusion, DSAI 2013, University of Vigo, Spain, November 13-15, 2013*, volume 27 of *Procedia Computer Science*, pages 57–66. Elsevier, 2013.
4. android.speech | android developers. <https://developer.android.google.cn/reference/android/speech/package-summary>, 2009.
5. Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race detection for event-driven mobile applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 326–336. ACM, 2014.
6. Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 316–325. ACM, 2014.
7. Jue Wang, Yanyan Jiang, Chang Xu, Qiwei Li, Tianxiao Gu, Jun Ma, Xiaoxing Ma, and Jian Lu. AATT+: effectively manifesting concurrency bugs in android apps. *Sci. Comput. Program.*, 163:1–18, 2018.
8. Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. Detecting event anomalies in event-based systems. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 25–37. ACM, 2015.
9. Yongjian Hu and Iulian Neamtiu. Static detection of event-based races in android apps. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 257–270. ACM, 2018.
10. Navid Salehnamadi, Abdulaziz Alshayban, Iftekhhar Ahmed, and Sam Malek. ER catcher: A static analysis framework for accurate and scalable event-race detection in android. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 324–335. IEEE, 2020.
11. Github - z3prover/z3: The z3 theorem prover. <https://github.com/z3prover/z3>, 2023.
12. Critical capabilities for cloud ai developer services. <https://www.gartner.com/en/documents/3999739>, 2021.
13. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014.
14. Download apk fast, free and safe on android. <https://apkpure.com/>, 2014.
15. Roma. <https://roma0216.github.io/Roma/>, 2023.

16. Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu. Effectively manifesting concurrency bugs in android apps. In Alex Potanin, Gail C. Murphy, Steve Reeves, and Jens Dietrich, editors, *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, pages 209–216. IEEE Computer Society, 2016.
17. Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 245–256. ACM, 2017.
18. Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical GUI testing of android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 269–280. IEEE / ACM, 2019.
19. Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. Combodroid: generating high-quality test inputs for android apps via use case combinations. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 469–480. ACM, 2020.
20. Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 153–164. ACM, 2020.
21. Steve Carr, Jean Mayo, and Ching-Kuang Shene. Race conditions: A case study. *J. Comput. Sci. Coll.*, 17(1):90105, oct 2001.
22. Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1(1):74–88, 1992.
23. Tanjila Farah, Rashed Shelim, Moniruz Zaman, Md Maruf Hassan, and Delwar Alam. Study of race condition: A privilege escalation vulnerability. 06 2017.
24. Kyung suk Lhee and Steve J. Chapin. Detection of file-based race conditions. *International Journal of Information Security*, 4:105–119, 2004.
25. Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Comput. Syst.*, 9(2):131–152, 1996.
26. Eugene Tsyklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.
27. Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada, June 18-21, 2000*, pages 219–232. ACM, 2000.
28. Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 90–96. ACM, 2001.
29. Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 237–252. ACM, 2003.
30. Nándor Licker and Andrew Rice. Detecting incorrect build rules. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1234–1244. IEEE / ACM, 2019.

31. Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 251–262. ACM, 2012.
32. Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 151–166. ACM, 2013.