When Voice Meets Touch: Conflict Analysis in Mobile Applications

Suwan Li, Lei Bu, *Member, IEEE*, Shangqing Liu, Guangdong Bai, *Member, IEEE*, Fuman Xie, Kai Chen, *Member, IEEE*, and Chang Yue

***Abstract*—The recent advancement of the automatic speech recognition (ASR) contributes to the voice user interface (VUI), which is broadly embedded into mobile apps. The VUI implemented on modern mobile operating systems like Android naturally involves multiple threads, and brings new race issues and challenges in defining and identifying them. Specifically, when the GUI and VUI (GV) actions both access to the same resource simultaneously, the data race named *GV-race* may occur. GV-race can lead to wrong behavior and even crashes. However, to the best of our knowledge, this problem has not been adequately studied. In this paper, we present the first study of GV-race in Android apps. However, the involvement of the VUI complicates the concurrency model, affects the temporal relationship and brings state space explosion in global analysis. To tackle these challenges, we firstly define *primitives* and their *happen-before* rules to abstract GV interaction patterns. Using these primitives, we are able to characterize and formally define GV-race. We then develop *Roma* (GV-race detector on mobile apps) to detect both app-level and system-level GV-race automatically. Through static program analysis, Roma extracts GV related call graphs for each pair of conflicting GV actions to reduce the state space, and generates a universal GV interaction graph using our pre-defined primitives. It encodes happen-before constraints to formally specify the *freeness of GV-race*, so that the detection of GV-race can be reduced to constraint solving with SMT solvers. We apply Roma to analyze 266 apps. Roma finds 52 apps with app-level GV-race and 56 apps with system-level GV-race. We confirm that 101 apps are true positives.**

***Index Terms*—Voice user interface, graphical user interface, data race, mobile applications.**

Received 13 May 2025; revised 8 January 2026; accepted 15 January 2026. Date of publication 26 January 2026; date of current version 17 March 2026. This work was supported by the National Natural Science Foundation of China under Grant 62232008, Grant 62172200, Grant 62572256, and Grant U24A20236. The work of Suwan Li, Lei Bu, and Shangqing Liu was supported in part by the National Key Research and Development Program of China under Grant 2024YFB2505604. The work of Guangdong Bai was supported in part by the CityUHK's Start-up Grant. The work of Kai Chen and Chang Yue was supported in part by the CAS Project for Young Scientists in Basic Research under Grant YSBR-118. Recommended for acceptance by S. Chattopadhyay. *(Corresponding author: Lei Bu.)*

Suwan Li, Lei Bu, and Shangqing Liu are with the State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: lisuwan@smail.nju.edu.cn; bulei@nju.edu.cn; shangqingliu@nju.edu.cn).

Guangdong Bai is with the Department of Computer Science, City University of Hong Kong, Hong Kong 999077, Hong Kong (e-mail: g.bai@cityu.edu.hk).

Fuman Xie is with the School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane 4072, Australia (e-mail: fuman.xie@uq.edu.au).

Kai Chen and Chang Yue are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China (e-mail: chenkai@iie.ac.cn; yuechang@iie.ac.cn).

Digital Object Identifier 10.1109/TSE.2026.3656691

I. Introduction

THE automatic speech recognition (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. The convenience of the VUI makes it widely used by developers for accessibility [1], [2] and welcomed by users. As reported by Statista [3], there are 31% of mobile users that use voice search at least once a week [4]. Despite the advantage brought by the integration of the GUI and the VUI, concerns about potential race conditions arise. Actions invoked through the VUI typically involve read or write operations on resources shared with the GUI. However, processing the voice inputs to identify the target resource often requires several seconds. This automatic speech recognition process is generally executed on a separate thread, leaving the main thread unprotected and thus susceptible to data races. We name this data race aroused by GUI and VUI (GV) actions as GV-race.

To illustrate a scenario of GV-race, consider the VUI framework of the Google VUI SDK [5]. The main thread records the user's speech and posts an ASR event, which is run by a background speech-to-text thread. The ASR process typically takes seconds, during which the main thread is idle. Under this VUI framework, the following GV-race may happen: the VUI action (e.g., say commands) happens before the GUI action (e.g., click a button), but the response to the VUI action arrives after the response to the GUI action. If the VUI action and the GUI action access the same resource, GV-race is detected.

GV-race deserves attention for the following three reasons. Firstly, GV-race is relatively *easy to trigger*. The long ASR latency of VUI actions leaves several seconds of unprotected execution time on the main thread, during which a conflicting GUI action can be easily injected. In contrast, most GUI actions respond almost instantaneously, making GUI-level data races much harder to reproduce or exploit. Secondly, GV-race is *difficult to mitigate*. Because VUI actions span multiple threads and callbacks, mitigating GV-race often requires coordinated modifications across several functions, further complicating the resolution. Finally, GV-race can result in *various consequences*.

Prior research has found that race conditions can result in various problems, including unexpected exit [6], [7], [8], responsiveness degradation [9] and battery drainage [10]. Specifically, GV-race can result in both usability issues and explicit errors. In navigation, food delivery, and translation apps, users may encounter unexpected destinations or incorrect translation results due to disruption of the GV action sequence. Because VUIs are frequently used by elderly individuals or people with disabilities, these users may struggle to verify or correct the results. Additionally, some instances of GV-race, particularly use-after-free races, can cause error messages or app crashes. In both cases, user experience is significantly compromised.

The widespread usage of the VUI, along with the significance of GV-race, require investigation on this emerging issue. Although numerous studies have investigated detecting event-driven races on GUI-based apps using static analysis [7], [8], [11], [12] or dynamic analysis [6], [13], concerns arising from the mixed presence of the GUI and the VUI have not been adequately studied. Specifically, the VUI and GV interaction are not characterized. GV interaction may involve **happen-in-the-middle behavior**. Unlike GUI actions, which typically respond immediately, VUI actions span multiple callbacks that can be interrupted or disabled by specific APIs. From the user's perspective, this makes VUI actions susceptible to interruption during execution. We refer to this phenomenon as "happen-in-the-middle" behavior. Existing GUI-level race detectors therefore focus on coarse-grained happen-before relationships at the action level, which is insufficient for accurately detecting GV-race. Instead, precise detection requires fine-grained modeling of GV interactions and detailed extraction of VUI implementations. Furthermore, analyzing GV-race at a global scale can result in **low efficiency**. Prior static-analysis-based race detectors capture all GUI actions and lifecycle events for global analysis. However, GV interactions introduce significantly more complex happen-before constraints, rendering such global approaches inefficient.

To address the limitations in characterizing VUI implementations and GV interactions, we analyze the detailed VUI implementation at the code level and propose specialized primitives for modeling GV interactions. We begin by examining VUI implementations in Android apps and categorizing them into synchronous and asynchronous forms. Our analysis reveals that VUI actions typically span multiple threads and callbacks, each responsible for distinct tasks. Accordingly, we decompose the VUI-related call graph into three parts, each representing one task. We then define a set of primitives tailored to GV interactions. In addition to primitives for threads, events, and traditional locks, we introduce new primitives to capture happen-in-the-middle behavior. Happen-before rules are further defined to specify execution ordering among these primitives. Building upon these primitives and rules, we formally define GV-race.

To adopt an efficient GV-race detection method, we propose *Roma* (GV-race detector on mobile apps). To manage complexity, data races involving more than two operations are decomposed into multiple scenarios, each containing only two operations, thereby reducing the scope of individual analyses. Accordingly, Roma adopts a pairwise detection strategy. Specifically, Roma treats a pair of GV actions as the fundamental detection unit, both at the source code level and at the primitive interaction level. In addition, the race detection results of GV action pairs with similar patterns will be saved to avoid repeated checks.

Overall, Roma contains three steps. Firstly, it *extracts GV-related call graphs* to ensure that the GV interaction behavior is accurately captured. Then, Roma *builds the GV interaction graph* using pre-defined primitives and happen-before rules to intuitively represent happen-before relationship during GV interaction at a fine-grained level. Finally, after *encoding happen-before constraints* based on the GV interaction graph, our problem is reduced to solving if all constraints can be satisfied by the SMT solver [14]. Despite app-level VUI actions, Roma also considers system-level VUI actions to enhance the issue scope.

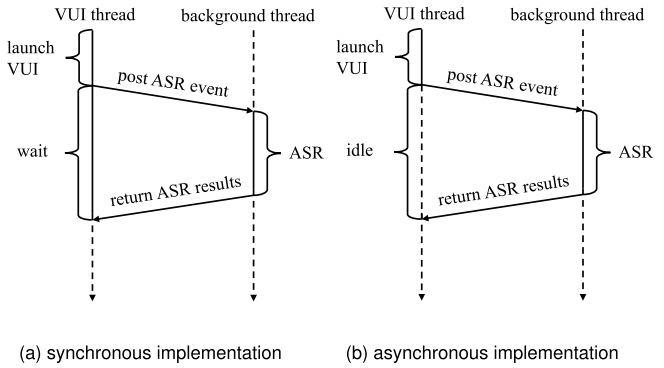
Finally, we conduct studies to investigate Roma's accuracy, efficiency and applicability. Roma analyzes 266 apps with different levels of VUI actions and discovers that GV-race broadly exists in GV co-access apps (containing in-parallel write-write or read-write on GV shared resources). Among them, Roma finds 108 apps with GV-race and we validate that 101 apps are true positives, achieving an accuracy rate of 93.5%. We also find that Roma can analyze over 95% of apps in 10 minutes. Our results emphasise the significance of GV-race detection and prevention.

Contributions: Contributions are summarized below.

- **Analysis and definition of GV-race.** We formally define GV-race and analyze its root cause. We discover that GV-race is aroused by the unprotected main thread during the VUI action and the unrestricted conflicting GUI actions. We also recommend strategies to mitigate GV-race. Developers are encouraged to add locks at specific positions to prevent unordered shared resource access.
- **Formal representation of GV interaction patterns.** We analyze the VUI framework of mainstream VUI SDKs and abstract their execution paradigms into two implementations. Towards complex VUI behavior and GV interactions, we define a new set of primitives and related happen-before rules to describe happen-in-the-middle behavior, along with traditional happen-before relationship.
- **Pairwise GV interaction graph based GV-race detection framework.** We design Roma, which introduces to use a pair of GV actions as the minimum detection unit and build the GV interaction graph to intuitively represent happen-before relationship. Happen-before constraints are encoded and solved for the detection of GV-race. Roma analyzes 266 apps, and discovers 101 real-world apps with GV-race. Our code and data are available at [15].

II. GV-Race

The VUI's time-consuming and multi-threaded features result in complex VUI-related races in Android apps. The VUI requires the automatic speech recognition process, which is mostly implemented by calling off-the-shelf Android SDKs. As a result, we analyze the VUI framework of mainstream Android SDKs and abstract their execution paradigms in Section II-A.



(a) synchronous implementation (b) asynchronous implementation

Fig. 1. VUI implementation patterns.

After that, we illustrate a real-world motivating example to help understand GV-race in Section II-B.

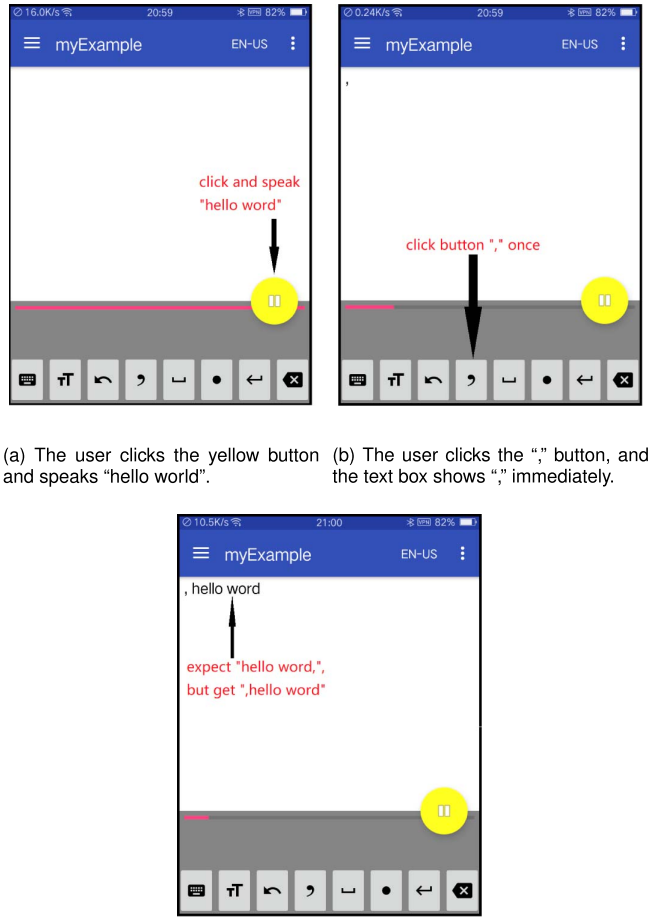
A. VUI Implementations in Android Apps

The Android system is essentially an event-driven multi-threaded system [16]. Different from normal multi-threaded systems, statements in the same thread of the Android system are not always executed in sequence because of the looper thread. The looper thread contains an event queue that can receive and execute events, where received events are executed in first-in first-out order. When an event is in execution, other events cannot interrupt the process. Correspondingly, there are non-looper threads, where statements are executed in order.

All UI related events are handled by the main thread, which is also a looper thread. Only light-weight events are recommended to be completely processed by the main thread to avoid the unresponsive interface or even crashes. Users' GUI actions, like clicking a button, are events handled by the main thread, which are mostly executed in milliseconds. The VUI actions are also converted into events handled by the main thread, but the time-consuming ASR process is typically processed by a background thread so that the main thread will not be blocked.

Typically, the implementation of the VUI involves at least two threads. Specifically, there is a VUI thread responsible for initiating the VUI, posting an ASR event and receiving the ASR results for further processing, and a background thread that conducts the ASR event and returns the ASR results. Based on the implementation of the VUI thread, we classify the VUI implementations into two types: synchronous and asynchronous. In the synchronous implementation (see Fig. 1(a)), the VUI thread waits until the ASR results return, which means that the VUI thread will not process other events from the initialization of the VUI to the return of the ASR results. In contrast, in the asynchronous implementation (see Fig. 1(b)), the VUI thread is idle after posting the ASR event to the background thread.

As part of our research, we investigated the Android VUI implementations of top-10 popular speech recognition service according to Gartner's study [17] in 2021, which rates the product or service for various language models. Six of them provide detailed documentation and sample apps. It is noteworthy that all of these SDKs (Alibaba, Baidu, Google, Huawei, Microsoft and Tencent) provide the asynchronous VUI implementations.



(a) The user clicks the yellow button and speaks "hello world". (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 GV-race.

Microsoft also provides a synchronous VUI implementation. Despite the variety of SDKs, they all conform to the two types of VUI implementations.

B. A Motivating Example of GV-Race

In this section, we provide an example to demonstrate GV-race. Fig. 2 shows the "Voice Notebook - continuous speech to text" app, which uses the Google SDK to implement the VUI. To use the VUI, the user clicks the yellow button and says commands (see Fig. 2(a)). After a few seconds, the ASR result is appended to the text box. By pressing the "," button, the "," will be immediately appended to the same text box (see Fig. 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 (see Fig. 2(c)). This is a common phenomenon because the VUI changes the temporal relationship: the main thread is idle for seconds during the ASR process, making it quite easy to insert a conflicting GUI action during the VUI action.

Fig. 3 shows the source code of related GV actions. When the user clicks on the button, the **onClick** function in class a is executed and it calls the R function in class MainActivity. The R function sets the class that defines the callback to receive the ASR event using the **setRecognitionListener** function and then

```
public class a implements View.OnClickListener {
    …
    public void onClick(View view) {//the VUI action is invoked
        …
        MainActivity.R();
        …
    }
}

public class MainActivity extends … implements RecognitionListener … {
    private SpeechRecognizer P;
    private EditText I;
    …
    public void R() {
      …
      this.P.setRecognitionListener(this);
      this.P.startListening(…);
      //post the ASR task to a background thread
      …
    }

    public void btnCommaClick(View view) {//the GUI action is invoked
      …
      this.I.setText(… + "," + …); //the GUI action writes to a textbox
      …
    }

    @Override
    public void onResults(Bundle bundle) { //acquire the ASR results
      …
      String str =
      bundle.getStringArrayList("results_recognition").get(0);
      …
      this.I.setText(… + str + …); //the VUI action writes to a textbox
      …
    }
}
```

Google VUI SDK

Fig. 3. Source code of the motivating example.

invokes the **startListening** function in the Google SDK. The **startListening** function sends the ASR event to the background thread, and returns immediately. When the user presses the "," button, the **btnCommaClick** function is executed and appends "," to the text box. A few seconds later, the background thread finishes the ASR event and sends the ASR results back. The main thread acquires the ASR results through the callback function **onResults** implemented in class MainActivity. The onResults function appends "hello word" to the text box. The VUI is invoked before the GUI, so the expected results should be "hello word," instead of ",hello word".

The VUI action involves multiple threads and takes much longer time to respond, complicating and facilitating the race. GV interactions may involve "happen-in-the-middle" behavior, where the VUI action can be interrupted by calling specific APIs. For example, if the **btnCommaClick** calls **stopListening** and **cancel** in the Google SDK before **setText** to prevent the VUI from returning the ASR results, **onResults** is not executed and GV-race is mitigated. Previous detectors focus on GUI level races, so they may fail to consider the happen-in-the-middle behavior in VUI actions. Without fine-grained analysis of GV interactions, there might be false positives. In order to analyze and detect GV-race systematically and accurately, we are required to define the GV interactions and GV-race formally.

III. Formalization and Definition

The Android system is a complex event-driven multi-threaded system. Detecting races in Android apps requires to analyze not only threads and locks, but also events in looper threads. The introduction of the VUI actions and GV interactions further complicates the analysis. Most GUI actions are handled by one event in one thread, but the VUI action involves at least two threads and multiple events. GV interactions involve happen-in-the-middle behavior, which requires fine-grained formalization and analysis to accurately detect GV-race.

To achieve accurate formalization, we work in two steps. Firstly, we extract detailed GV implementations at the code level. We introduce terms to describe GV implementation and interaction related functions or call graphs in Section III-A. Secondly, we abstract the GV interactions by defining primitives and corresponding happen-before rules. Primitives represent threads, events, locks and happen-in-the-middle behavior that may affect the temporal relationship (see Section III-B). Happen-before rules describe the execution sequence between specific primitives according to the system implementation and code semantics (see Section III-C). Using these primitives and rules, we formally define GV-race in Section III-D.

A. GV-Race Related Terms at the Source Code

GV-race arises on the premise of in-parallel GV operations on GV shared resources, and at least one of them is a write operation. As we focus on VUI related races, we only extract codes of VUI actions and GUI actions that share the same resource with VUI actions. Specifically, we extract the VUI call graph that contains call relationships from launching the VUI to the write or read operations, and the GUI call graph that contains call relationships from the GUI related callbacks (such as "onClick") to the write or read operations that access the same resource as the VUI actions.

Unlike the GUI call graph, the VUI call graph is more complex as it involves multiple tasks, including launching and recording the voice, conducting the ASR and processing the ASR results. Different tasks are handled by different developers. For example, the ASR process is mostly implemented using SDKs, but the ASR results are handled by app developers. Due to these reasons, we split the VUI related call graph into three parts to represent different tasks: VUI launch related call graph, VUI ASR related call graph and VUI source related call graph. Below, we define terms of important functions and call relationships at the source code level.

- **VUI source**: the callback to get ASR results, e.g. **onResults**.
- **GUI source**: the callback related to a GUI action, including lifecycle related callbacks, e.g. **btnCommaClick**, onPause.
- **Sink (VUI / GUI sink)**: the high-level operations that access the GV shared resource, e.g. **setText**.
- $\boldsymbol{f}$ **related call graph**: represented as $CG_{\boldsymbol{f}} = \{F, R, \boldsymbol{f}\}$.
  - F is the set of functions.
  - R is the set of call relationships. $R = \{f_0 \rightarrow [f_1, f_2, \ldots, f_n] | f_0 \ldots f_n \in F\}$. $f_0 \rightarrow [f_1, f_2, \ldots, f_n]$ means that the caller f_0 calls callees $f_1 \ldots f_n$. The sequence of callees is the same as that in the source code.
  - $\boldsymbol{f} \in F$ and $\boldsymbol{f}$ is the entry function of $CG_{\boldsymbol{f}}$.
- **GUI source related call graph**: the call relationships of threads, events, locks or GV interactions related functions

from GUI source to GUI sink, e.g. $CG_{\text{btnCommaClick}} = \{\{\text{btnCommaClick, setText}\}, \{\text{btnCommaClick} \rightarrow [\text{setText}]\}, \text{btnCommaClick}\}$.

- **VUI launch related call graph**: the call relationships of threads, events, locks or GV interactions related functions from the launch of the VUI to the start of the ASR event, e.g. $CG_{\text{onClick}} = \{\{\text{onClick, R, startListening}\}, \{\text{onClick} \rightarrow [\text{R}], \text{R} \rightarrow [\text{startListening}]\}, \text{onClick}\}$.
- **VUI ASR related call graph**: the call relationships from the start of the ASR to VUI source, e.g., from startListening to onResults. It can be achieved from the Google VUI SDK.
- **VUI source related call graph**: the call relationships of threads, events, locks or GV interactions related functions from VUI source to VUI sink, e.g. $CG_{\text{onResults}} = \{\{\text{onResults, setText}\}, \{\text{onResults} \rightarrow [\text{setText}]\}, \text{onResults}\}$.
- **GV related call graphs**: a joint of the four call graphs related to a pair of GV actions.

With these terms, we can focus on details related to GV implementation and interaction. However, the call graph represents call relationship rather than temporal relationship, which is not intuitive for GV-race analysis. Therefore, we propose to introduce primitives and corresponding happen-before rules to represent happen-before relationship during GV interactions.

B. GV Interaction Related Primitives

Although it is possible to directly detect GV-race on the call graph, the call graph primarily represents call relationships rather than happen-before relationships, which makes the analysis less intuitive. To address this limitation, we manually define primitives that capture the core behaviors affecting temporal relationships, including traditional behaviors and GV-interaction-specific behaviors. We include traditional primitives related to threads, events, and locks, such as thread creation and event posting. Different from prior work, we additionally introduce fine-grained primitives to capture the happen-in-the-middle behavior during GV interactions or the disabling of GV actions. Based on our observations, we identify two scenarios in which these primitives are applicable. The first scenario involves interrupting a VUI action during its execution by invoking specific APIs. These APIs can disable the callback events that handle VUI results (the VUI source), as well as the associated VUI sinks. The second scenario involves preventing the execution of GUI-related callback events by modifying the layout. In both cases, the common effect is the prevention of another event from being executed. Our primitives are classified into four groups as presented in Table I.

- **Thread-related primitives**. Thread-related primitives represent operations to create, join, start, or end a thread.
- **Event-related primitives**. The involvement of the looper thread and event queue requires event-related primitives. They represent operations to post, start or end an event.

TABLE I
GV INTERACTION RELATED PRIMITIVES

Type	Primitive	Definition
Thread -related	threadBegin(t)	start the thread t
	threadEnd(t)	end the thread t
	fork(t, t')	create the thread t' by thread t
	join(t', t)	the thread t' joins to thread t
	executeEvent(t)	start executing events
Event -related	postEvent(t, p, t')	post an event p to thread t' by thread t
	eventBegin(t, p)	start the event p in the thread t
	eventEnd(t, p)	end the event p in the thread t
Lock -related	disable(t, p)	disable executing the event p in the thread t
	enable(t, p)	enable executing the event p in the thread t
	lock(t, l)	acquire the lock l in thread t
	unlock(t, l)	release the lock l in thread t
GV -related	SpeechRecognition(t)	thread t transcribes the speech to text
	VUIsink(t)	the VUI read or VUI write in thread t
	GUIsink(t)	the GUI write in thread t

- **Lock-related primitives**. Traditional locks and operations to disable VUI or GUI actions are represented by lock-related primitives.
- **GV-related primitives**. GV-related primitives represent the ASR process, VUI sink, and GUI sink.

These primitives form a united language to describe the behavior that may impact the temporal relationship during GV interactions, regardless of the source code and SDKs. By mapping the source code to the primitives, we can analyze the temporal relationship during GV interactions more directly.

C. Happen-Before Rules

GV-race happens because of the disorder of temporal relationship. To validate the satisfaction of GV-race under a given GV interaction scenario, we are required to analyze the temporal relationship during GV interactions. We focus especially on happen-before relationship, which describes the relationship of an statement executing before another statement. For example, the statements in a non-looper thread always execute one by one. Based on different locations and sources, we define three categories of happen-before rules, namely sequence within events or threads (SWE), sequence between events in the same queue (SBE), and sequence determined by semantics (SDS). For the purpose of our discussion, we define the following expressions:

- $\frac{condition}{constraint}$ represents that once the condition $condition$ is satisfied, we can infer the happen-before constraint $constraint$.
- $a_i \leq a_j$ represents that a_i happens before a_j.

- $event(a_i)$ represents the event id of a_i.
- $thread(a_i)$ represents the thread id of a_i.
- "_" is a placeholder that represents any thread or event ids.

Sequence within events or threads (SWE) represents happen-before rules of primitives in the same event or thread. As intra-event primitives are executed sequentially, we introduce Rule 1 : if the primitive a_i and the primitive a_j are in the same task, and the primitive a_i is written before the primitive a_j, there is a happen-before constraint $a_i \leq a_j$.

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

In a looper thread, events only begin to execute after the event queue starts. As shown in Rule 2, the looper thread t starts the event queue with the primitive executeEvent (t). After that, events posted to this event queue labeled with eventBegin $(t, \_)$ can start to execute.

$$\text{executeEvent}(t) \leq \text{eventBegin}(t, \_) \tag{2}$$

In a non-looper thread or before the event queue starts, primitives are executed sequentially (see Rule 3). Given a list of primitives $\{a_1, ..., a_n\}$ in the same thread, they are ordered by the program sequence and a_1 represents the first primitive in this thread. If these primitives do not contain executeEvent or only the last primitive is executeEvent, there is a happen-before relationship between any a_i and a_j pairs on the premise that $i < j$.

$$\frac{(\text{executeEvent} \notin \{a_1, ..., a_n\} \vee a_n = \text{executeEvent}) \wedge (i < j \leq n) \wedge (thread(a_i) = thread(a_j))}{a_i \leq a_j} \tag{3}$$

Sequence between events in the same queue (SBE) represents happen-before rules of primitives in different events. These rules reflect the fact that events in the same queue are executed in first-in, first-out order. If the event p_1 is posted to the same thread before the event p_2, then p_1 ends before p_2 starts (see Rule 4).

$$\frac{\text{postEvent}(\_, p_1, t) \leq \text{postEvent}(\_, p_2, t)}{\text{eventEnd}(t, p_1) \leq \text{eventBegin}(t, p_2)} \tag{4}$$

If we already know the happen-before constraint between two primitives from different tasks in the same thread, we can predict the happen-before constraint between these two tasks. Specifically, as shown in Rule 5, if there is an existing relationship that a_m from event p_1 happens before a_n from event p_2, and a_m, a_n are in the same thread, then the task p_1 ends before the task p_2 starts.

$$\frac{event(a_m) = (t, p_1) \wedge event(a_n) = (t, p_2) \wedge a_m \leq a_n}{\text{eventEnd}(t, p_1) \leq \text{eventBegin}(t, p_2)} \tag{5}$$

Sequence determined by semantics (SDS) represents happen-before rules determined by semantics. An event is posted before it starts to execute as summarized by Rule 6. The post of the event p to the thread t' happens before the event p starts.

$$\text{postEvent}(t, p, t') \leq \text{eventBegin}(t', p) \tag{6}$$

Rule 7 represents that a thread is created before it starts. The thread t forks a new thread t', and then the thread t' starts to execute.

$$\text{fork}(t, t') \leq \text{threadBegin}(t') \tag{7}$$

If a thread is forked by another thread, it may join to its parent thread after it ends. Rule 8 represents that the thread t' ends before joining to thread t.

$$\text{threadEnd}(t') \leq \text{join}(t', t) \tag{8}$$

There are also rules related to disabling and enabling behavior due to complex GV interaction. We identify two scenarios in which this rule is applied: (1) happen-in-the-middle behavior, and (2) modifying the layout to disable a GUI-related callback. In both cases, the common effect is the prevention of an event from being executed. Since an event can only be executed before it is disabled or after it is enabled, we propose the Rule 9, which means that the post of the event p to the thread t happens before disabling the event p, or after enabling the event p.

$$\begin{aligned}\text{enable}(t, p) \leq \text{postEvent}(\_, p, t) \vee \text{postEvent}(\_, p, t) \\ \leq \text{disable}(t, p)\end{aligned} \tag{9}$$

Rule 10 represents the happen-before rule introduced by traditional locks. When a lock is taken by one thread, other threads can only acquire this lock after it is released. As shown in Rule 10, either the thread t releases the lock before thread t' acquires it, or thread t' releases the lock before thread t acquires it.

$$\text{unlock}(t, l) \leq \text{lock}(t', l) \vee \text{unlock}(t', l) \leq \text{lock}(t, l) \tag{10}$$

Rule 11 represents that the happen-before constraint is transitive. If there are happen-before constraints that a_i happens before a_k and a_k happens before a_j, then a_i also happens before a_j.

$$\frac{a_i \leq a_k \wedge a_k \leq a_j}{a_i \leq a_j} \tag{11}$$

Happen-before constraints are added on primitives according to happen-before rules to specify the temporal relationship in the GV interaction scenario. Given primitives and happen-before rules, we are allowed to define and analyze GV-race systematically and formally.

D. GV-Race Definition

Given primitives in Section III-B and happen-before rules in Section III-C, we formally define GV-race in Definition 1.

Definition 1: Given a GUI action from postEvent(_, $click_{GUI}, t_0$) to GUIsink (t_1) and a VUI action from postEvent $(\_, click_{VUI}, t_0)$ to VUIsink (t_2), GUIsink (t_1) and VUIsink (t_2) access the same resource. If postEvent $(\_, click_{VUI}, t_0) \leq$ GUIsink $(t_1) \wedge$ postEvent $(\_, click_{GUI}, t_0) \leq$ VUIsink (t_2) is satisfied, **GV-race** happens.

As the response time of the VUI is much longer than that of the GUI, the GUI-before-VUI race hardly happens. Without loss of generality, we focus on the VUI-before-GUI GV-race.

Definition 2: Given a GUI action from postEvent(_, $click_{GUI}, t_0$) to GUIsink (t_1) and a VUI action from postEvent

$(\_, click_{VUI}, t_0)$ to VUIsink (t_2), GUIsink (t_1) and VUIsink (t_2) access the same resource. If the **GV-race constraint** postEvent $(\_, click_{VUI}, t_0) \leq$ postEvent $(\_, click_{GUI}, t_0) \wedge$ GUIsink $(t_1) \leq$ VUIsink (t_2) is satisfied, **VUI-before-GUI GV-race** happens.

IV. Pairwise GV Interaction Graph Based GV-Race Detection

The VUI complicates the race conditions in Android apps because the VUI involves multiple threads and events, and GV interactions involve happen-in-the-middle behavior. In the previous section, we introduce primitives and happen-before rules to help represent the GV interaction and GV-race. However, automatic GV-race detection still faces challenges of effectively abstracting GV interaction pattern from the source code and efficiently analyzing GV-race. In this section, we introduce Roma (GV-race detector on mobile apps) to handle these issues. Roma incrementally constructs the GV interaction graph by exploring the pre-extracted GV related call graphs. It adopts a lightweight pairwise detection framework to reduce analysis time.

A. GV Interaction Graph and Detection Framework

As we explained in Section III-A and Section III-B, call graph represents call relationship rather than temporal relationship, making it difficult to directly analyze the satisfactory of GV-race. To abstract the temporal relationship in GV interaction, we introduce primitives and happen-before rules. In the detection scope, we define the **GV interaction graph** to represent the happen-before relationship during GV interactions using primitives.

Definition 3: The GV interaction graph G represents the GV interaction at the primitive level. It is defined as $G = (V, E)$, where:

- V is the set of nodes. A node represents a primitive.
- E is the set of edges. $E = \{(u, v) | u \in V, v \in V\}$. The edge (u, v) means that u happens before v (i.e., $u \leq v$).

Global detection of GV-race suffers from efficiency issues due to the complexity of GV interactions, which introduce numerous constraints that must be analyzed when checking GV-race satisfaction. As we focus on the detection of data race introduced by the VUI in Android apps, we only retain GUI actions that interact with the VUI for efficient analysis. In practice, a data race involving more than two operations can be decomposed into multiple pairs, each consisting of two operations without a predefined happen-before relationship. The constraint checking on small-scaled scenario can greatly save time. Building on this observation, we adopt a pair of conflicting GV actions as the fundamental detection unit, and save the checking results of each pair of GV actions to avoid repeated checking. A VUI action can interact with multiple GUI actions, and their interaction patterns are likely to be similar, so previous analysis results will be used for GV action pairs with similar patterns.

Based on these ideas, we propose Roma, a lightweight pairwise GV interaction graph based GV-race detection framework. Fig. 4 shows its framework. Roma starts by extracting GV related call graphs (Step 1), based on which, Roma builds the GV interaction graph (Step 2) for each pair of conflicting GV actions. Finally, happen-before constraints are encoded and solved (Step 3) by the SMT solver [14] for GV-race detection. These steps are detailed below.

Step 1: Extract GV related call graphs. Given an apk file, Roma uses off-the-shelf tools to build the entire call graph. Through static analysis, Roma locates pairwise conflicting GV actions and extracts their GV related call graphs.

Step 2: Build GV interaction graph. Step 2 takes the GV related call graph as the input. By traversing the GV related call graph, mapping functions to primitives and adding the happen-before constraints, Roma builds the GV interaction graph for a pair of potentially conflicting *GV actions*.

Step 3: Encode and solve the happen-before constraints. Step 3 takes the GV interaction graph as the input. If the GV interaction graph is checked before, the previous result is reused. Otherwise, Roma encodes all happen-before constraints according to happen-before rules in Section III-C on the GV interaction graph. The GV-race constraint and these happen-before constraints are solved by the SMT solver [14] to check the satisfaction of GV-race. Finally, the checking result is recorded for reuse.

B. Extract GV Related Call Graphs

To detect GV-race efficiently, Roma only focuses on VUI actions and GUI actions that interact with VUI actions. Given the entire call graph, Roma automatically locates VUI actions and their VUI related call graph, and then finds potentially conflicting GUI actions and their GUI related call graph through alias analysis and call graph exploration. These extracted call graphs of potentially conflicting GV action pairs are called the GV related call graphs.

This process is divided into six steps: Step (1). locate VUI sources, Step (2). extract VUI launch related call graphs, Step (3). extract VUI ASR related call graphs, Step (4). extract VUI source related call graphs, Step (5). locate possible GUI sinks and Step (6). extract GUI source related call graphs. Fig. 5 explains the process.

Step (1). locate VUI sources. Roma locates functions that represent VUI sources, such as the **onResults** function.

Step (2). extract VUI launch related call graphs. Roma locates the corresponding function to start the ASR, such as the **startListening** function. Starting from that function, Roma traverses the call graph backward until a callback related to the user's action is found. Call relationships from that callback to the function that starts the ASR form the VUI launch related call graph.

Step (3). extract VUI ASR related call graphs. VUI ASR related call graphs contain call relationships from the function that starts the ASR found in **Step (2)** to the VUI source located in **Step (1)**. If the source code of the ASR launch function is available, this call graph can be obtained by off-the-shelf tools. If the VUI is implemented by VUI SDKs, the call graph built by off-the-shelf tools may miss these call relationships.

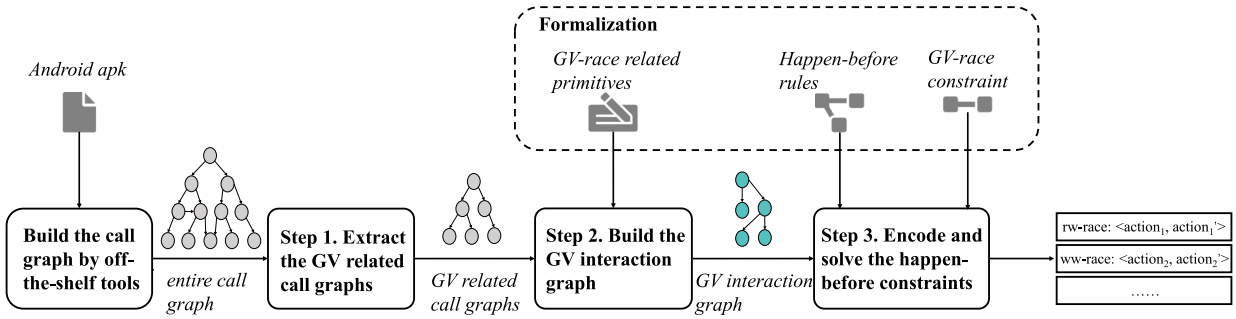


Fig. 4. The overall framework of Roma.

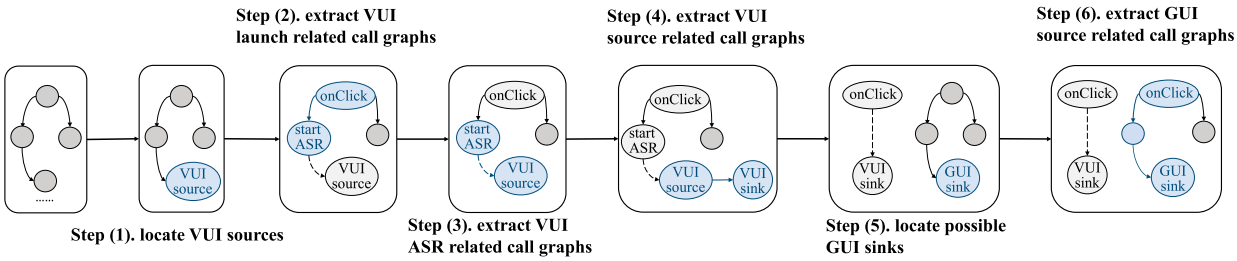


Fig. 5. The process to extract GV related call graphs.

Prior race detection techniques may fail to capture call relationships defined in external libraries [12] without domain-specific knowledge. In Roma, we incorporate the call relationships of six VUI SDKs (see Section II-A). To obtain their implementations, we download the official APK files provided by the respective companies, decompile them using off-the-shelf tools [18], and identify the corresponding APIs along with their implementations. We then retain the functions related to threads, events, and locks, along with their call relationships, tracing from the function that initiates the ASR task to the VUI source. Once the call graph for a given VUI SDK is constructed, it can be reused for other applications employing the same SDK, thereby minimizing extra effort.

Step (4). extract VUI source related call graphs. Starting from the VUI source, Roma traverses through the call graph forward until statements that represent high-level write or read operations are found. These statements are recorded as VUI sinks. Call relationships from the VUI source to one VUI sink form a VUI source related call graph.

Step (5). locate possible GUI sinks. Roma locates the object accessed by one VUI sink in **Step (4)**. Then, it uses alias analysis to identify statements that write to the same object, and records them as GUI sinks.

Step (6). extract GUI source related call graphs. Starting from a GUI sink, Roma traverses through the call graph backward to find callbacks related to the user's actions. These callbacks are GUI sources. Call relationships from a GUI source to the GUI sink form a GUI source related call graph.

To identify the sources, sinks, and other GV-related functions in the call graph, we construct a comprehensive list based on the Android SDK and six VUI SDKs. Most GUI sources correspond to callbacks of GUI actions or lifecycle events, such as "onClick", "onTouch", "onPause", and "onResume". Another part of GUI sources are defined by developers and collected by automatically extracting from the layout files. VUI sources and related functions are summarized from these six VUI SDKs; for instance, "onResults" is defined as a source function in the Google VUI SDK. Sinks represent high-level read and write operations. A subset of them consists of Android SDK APIs that access widgets, such as "setText" and "getText" that operate on text boxes. Another subset of sinks represents free and use operations. A free operation is identified when an object is set to null or released through specific SDK-defined APIs, while the corresponding use operation refers to any statement that accesses this object. VUI related functions are reusable across apps that employ these six VUI SDKs, whereas GUI sources and all sinks are generally reusable across Android apps.

A pair of VUI source and GUI source extracted in **Step (1)** and **Step (6)** is the potentially conflicting pairwise GV actions and the smallest analysis unit. Their corresponding four call graphs form the GV related call graph, which is used to build the GV interaction graph for GV-race detection.

C. Build GV Interaction Graph

The GV related call graph contains call relationship of potential conflicting GV action pairs, but it cannot intuitively reflect the temporal relationship during GV interactions. That motivates us to construct the GV interaction graph, with nodes of primitives and edges of happen-before relationship. Based on the semantics of the primitives defined in Section III-A, we summarized a list of corresponding functions or source code locations to map to these primitives. Roma builds the GV interaction graph automatically by mapping functions to

Algorithm 1 Build the GV interaction graph

Input: Call relationships in GV related call graphs cgs, mappings from functions to primitives map
Output: The GV interaction graph: G

```
1: for cg in cgs do
2:     for funct in cg.entry() do
3:         ProcessFunc(funct, cg)
4:     end for
5: end for
6: return G
1: function ProcessFunc(funct, cg)
2:     prim ← map.getPrim(funct)
3:     primName ← prim.getName() ▷ map the funct to the prim, and get the primitive name primName
4:     G.addNode(prim)
5:     if primName == "fork" then
6:         threadBegin = graph.addNode("threadBegin")
7:         graph.addEdge(prim, threadBegin)  ▷ add thread creation edges, see Rule 7
8:         for nextF in cg.callees(funct) do
9:             ProcessFunc(nextF, cg)
10:        end for
11:        G.addNode("threadEnd")
12:    else if primName == "postEvent" then
13:        prePrim = graph.addNode("eventBegin")
14:        graph.addEdge(prim, prePrim)  ▷ add event creation edges, see Rule 6
15:        for nextF in cg.callees(funct) do
16:            curPrim =ProcessFunc(nextF, cg)
17:            graph.addEdge(prePrim, curPrim)  ▷ add intra-event edges, see Rule 1
18:            prePrim = curPrim
19:        end for
20:        curPrim = graph.addNode("eventEnd")
21:        graph.addEdge(prePrim, curPrim)  ▷ add intra-event edges, see Rule 1
22:    else
23:        G.addNode(prim)
24:        for nextF in cg.callees(funct) do
25:            ProcessFunc(nextF, cg)
26:        end for
27:    end if
28:    return prim
29: end function
```

primitives (i.e., nodes) and adding happen-before constraints (i.e., edges) during the depth-first traversal of call graphs. Edges representing happen-before sequence consistent with the traversal sequence, such as intra-event rules (see Rule 1) and creation-related inter-event rules (see Rule 6-7), are added to the GV interaction graph in this step. Algorithm 1 shows the process to build the GV interaction graph from the GV related call graph.

- From the entry $funct$ in each call graph cg, we start to add primitives by the ProcessFunc function (lines 2-3).
- In the ProcessFunc function, we map $funct$ to the corresponding primitive $prim$ and add $prim$ to the GV interaction graph G (ProcessFunc, line 2-4). If $prim$ is a fork primitive, we add the threadBegin primitive to the GV interaction graph G, and an edge from the fork primitive to the threadBegin primitive (see Rule 7). Then, we call the ProcessFunc function to process the callees. After that, we add a threadEnd primitive to the GV interaction graph G (ProcessFunc, line 5-11). If $prim$ is a postEvent primitive, the procedure is similar (ProcessFunc, lines 12-21). In other cases, we add the corresponding $prim$ to the GV interaction graph G and call ProcessFunc to process the rest call graph (ProcessFunc, lines 23-27). Finally, the ProcessFunc function returns the corresponding primitive $prim$ of the function $funct$ (ProcessFunc, lines 28).
- After the ProcessFunc function returns, we get the GV interaction graph G (line 4).

D. Encode and Solve the Happen-Before Constraints

The GV interaction graph represents the interaction patterns of potential conflicting GV action pairs. To check the satisfaction of GV-race, happen-before constraints represented by edges in the GV interaction graph must be satisfied. However, edges in the GV interaction graph do not cover all the constraints as only happen-before constraints consistent with the traversal sequence are added to the GV interaction graph in Step 2. Other happen-before constraints, such as constraints deduced by Rule 10 and Rule 9, should be encoded and satisfied to detect GV-race. If the GV-race constraint and all happen-before constraints are satisfied, GV-race is detected.

Specifically, Roma transfers the happen-before constraints to less-than constraints and encodes them to the constraint set. Roma generates a variable for each node in the GV interaction graph. Given an existing edge (a_i, a_j), if the variable v_i represents the node a_i and the variable v_j represents the node a_j, Roma encodes the less-than constraint $v_i < v_j$. For complex happen-before rules that involve more than two primitives, such as the disable-enable rule (see Rule 9) and the lock-unlock rule (see Rule 10), Roma replaces the primitive with the corresponding variable and the happen-before relationship $\leq$ with the less-than relationship $<$ to generate the constraint. Algorithm 2 shows the process to encode happen-before constraints. It works as follows:

1) The GV-race constraint is encoded and added to C (line 1). The edges in the GV interaction graph G are also encoded as constraints (line 2-4).
2) We check each pair of a_i and a_j, where v_i and v_j are their corresponding variables, respectively. If a_i and a_j are in the same thread and the precondition of the SWE rules is satisfied, we add the constraint $v_i < v_j$ (line 7-9). If they are in different events and the precondition of the SBE rules is satisfied, the constraint $v_i < v_j$ is added (line 10-12). The relationship of a_i and a_j is also checked according to the SDS rules (line 13-15). If there are locks (including enable and disable), the constraints are added

Algorithm 2 Encode Constraints on the GV interaction graph

Input: The GV interaction graph G, happen-before rules swe, sbe and sds, the GV-race constraint $GV\_Constraint$
Output: The encoded happen-before constraints C.

1: $C \leftarrow GV\_Constraint$ ▷ prepare the GV-raceconstraint
2: **for** (a_i, a_j) in G.**getEdges**() **do**
3: $C \leftarrow C \wedge (v_i < v_j)$
4: **end for**
5: **while** C.**updated()** **do**
6: **for** a_i, a_j in $graph$.**getNodes**() **do**
7: **if** **thread**(a_i) == **thread**(a_j) && swe.**getPre**(a_i, a_j) is True **then** ▷ add SWE
8: $C \leftarrow C \wedge (v_i < v_j)$
9: **end if**
10: **if** **event**(a_i) != **event**(a_j) && sbe.**getPre**(a_i, a_j) is True **then** ▷ add SBE
11: $C \leftarrow C \wedge (v_i < v_j)$
12: **end if**
13: **if** sds.**getPre**(a_i, a_j) is True **then** ▷ add SDS
14: $C \leftarrow C \wedge (v_i < v_j)$
15: **end if**
16: **end for**
17: **if** $graph$.**containLocks**() **then** ▷ add SDS (locks)
18: $C \leftarrow C \wedge graph$.**getLockConstraints**()
19: **end if**
20: **end while**
21: **return** C

according to rule 9 and rule 10 (line 17-19). This process continues until C no longer changes (line 5-19).

3) Finally, the constraint C is returned (line 19).

The encoded constraints are solved by SMT solvers. If the constraints can be satisfied, GV-race happens. The proof of this theorem is shown below. The solver's results and the GV interaction graph are saved to avoid double checking.

Theorem 1: Given a GV interaction graph G and its happen-before constraints C encoded by Algorithm 2, if C is satisfied, GV-race happens. The proof of this theorem is shown below.

Proof: In each round, each pair of a_i and a_j in the GV interaction graph G is checked to add happen-before constraints $hbc = a_i \leq a_j$. Happen-before constraints hbc involving more than two primitives (see Rule 9 and Rule 10) are also added in each round. The loop ends until the happen-before constraints C no longer changes. As a result, all happen-before constraints are added to C iteratively under the GV-race constraint based on the GV interaction graph G. Finally, the constraint is $C = GV\_Constraint \wedge hbc_1 \wedge hbc_2 ... \wedge hbc_n$. The result that satisfies constraint C must satisfy the GV-race constraint. Therefore, if C is satisfied, GV-race happens. □

Fig. 6 shows an example to build the GV interaction graph from GV related call graphs, and encode constraints on the GV interaction graph. In Step 2, only specific functions are mapped to nodes represented by primitives, while unnecessary functions and call relationships are ignored. For example, the GUI source btnCommaClick is mapped to the primitive postEvent $(t_0, click_{GUI}, t_0)$. Edges in the GV interaction graph (marked by blue lines) are incrementally added based on happen-before rules. For example, the black edge from postEvent $(t_0, click_{VUI}, t_0)$ to eventBegin $(t_0, click_{VUI})$ is added based on Rule 6 in the SDS rules. The black edge from eventBegin $(t_0, click_{VUI})$ to fork (t_0, t_1) is added based on Rule 1 in the SWE rules.

In Step 3, more constraints are encoded based on the GV interaction graph. The blue edges represent the newly added happen-before constraints, while the purple edges represent the GV-race constraint. After adding the GV-race constraint, the precondition of Rule 4 is satisfied, so a new SBT edge from eventEnd $(t_0, click_{VUI})$ to eventBegin $(t_0, click_{GUI})$ is added. When the constraint set no longer changes, it is solved by SMT solvers to detect GV-race. In this case, constraints can be satisfied, so this app is detected with GV-race.

V. Evaluation

Our evaluation aims to answer the following research questions.

RQ1: Accuracy. How is the accuracy performance of Roma compared with related tools?
RQ2: Efficiency. How much time does Roma require to analyze an app? How effective is our pairwise detection strategy in improving efficiency?
RQ3: Ablation study. How do our formalization contribute to GV-race detection?
RQ4: Applicability. Is Roma applicable to apps implemented by other VUI SDKs?

A. Setup

Dataset: To the best of our knowledge, there is no off-the-shelf dataset of mobile apps that support the VUI. We crawl Android apps from apkpure [19] to acquire a dataset. Apkpure is a website that provides APK files for download, including both current and previous versions. Popular apps available in official app stores can be found on this platform. The initial dataset consists of 10,000 apps spanning all categories. Since not all apps support the VUI functionality or implement it using the six VUI SDKs, we identy VUI-enabled apps based on their features and classify them into three datasets according to the sources of their VUI actions. They are the GV-in-app dataset with VUI actions inside the tested app, the GV-between-apps dataset with VUI actions from another app and the system-level dataset with system-level VUI actions.

1) *GV-in-app dataset.* The VUI of apps in this dataset is implemented inside the tested apps. As Google VUI SDK is the most popular VUI SDK for VUI implementation, this dataset contains apps with VUIs that are implemented by the Google VUI SDK. We identify these apps by detecting the use of VUI-related APIs.
2) *GV-between-apps dataset.* Apps in this dataset start an intent to use the VUI service provided by other apps. For example, the "Speech Recognition & Synthesis" app offers the intent to use the Google Speech Recognition

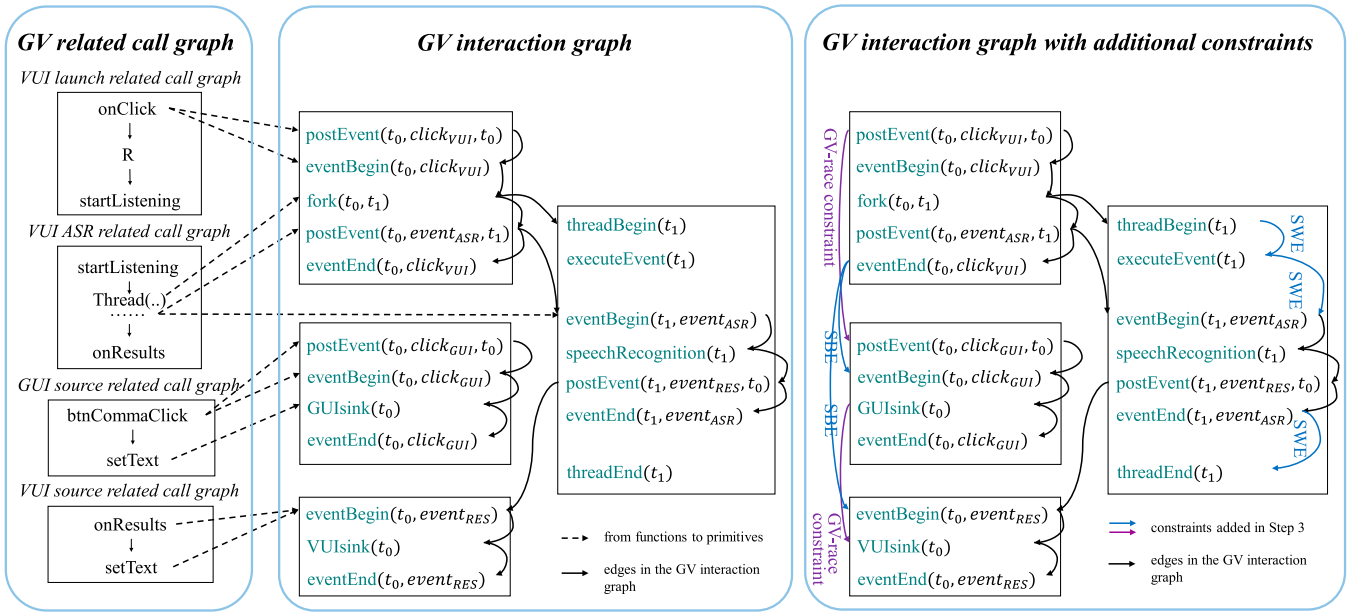


Fig. 6. Build the GV interaction graph from GV related call graphs and encode constraints.

service. We identify these apps by detecting the use of the corresponding intents.

3) *System-level dataset.* Apps in this dataset may not implement or call the VUI, but they can be accessed by system-level VUI actions. By installing the Voice Access app and enabling it in the settings, system-level VUI actions are introduced on running apps. The system-level VUI actions can read or write any widgets on the current activity based on voice commands. These apps do not necessarily implement VUI functionality; therefore, this dataset comprises the top 70 most popular apps.

After running FlowDroid [20] to construct the graphs on this dataset, we filter out apps with these characteristics: 1. apps that cannot be correctly analyzed by FlowDroid. 2. apps that cannot be analyzed by FlowDroid in 30 minutes. 3. apps whose VUI source is not in the call graph. Finally, we get 122 apps in the GV-in-app dataset, 88 apps in the GV-between-apps dataset and 56 apps in the system-level dataset for evaluation. Approximately 96.2% of these apps have been downloaded more than 100,000 times, including popular apps from well-known companies such as Google, Amazon, and NVIDIA. Our analysis is conducted on these 266 apps.

Baselines: We compare Roma with one publicly available race detection tool for Android apps, as described below:

- nAdroid [8]. nAdroid is an event-driven race detection tool that focuses on use-after-free (UAF) vulnerabilities in Android apps. It introduces the threadification technique, which converts ordering relationships between event callbacks into ordering relationships between threads. The tool is capable of modeling lifecycle callbacks, GUI callbacks, system events, as well as Handler and AsyncTask. After threadifcation, nAdroid applies state-of-the-art thread-based race detection techniques to identify potential races. To reduce false positives, it employs sound and unsound filters to prune benign or spurious races. Notably, nAdroid does not model or analyze locks, as locks are not effective in preventing UAF races.
- ER Catcher [12]. ER Catcher is a flow-, context-, and thread-sensitive static analysis framework designed to detect event-driven races in Android apps. It introduces the Concurrency-aware Summary Function (CSF) to model the concurrent behavior of methods in both Android apps and libraries. ER Catcher constructs a Context- and Concurrency-aware Call Graph (C^3G) based on the call graph generated by FlowDroid [20]. Finally, it employs a Static Vector Clock to establish happen-before relationships according to three principles that define ordering rules within the call graph and FIFO threads.

Implementation: We implement Roma to detect GV-race in Android apps. Roma uses the SPARK algorithm provided by FlowDroid [20] to construct the entire call graph. The0x000a0GV related call graph is extracted based on this whole call graph. Roma supports detecting the VUIs implemented by six SDKs mentioned in Section II-A. We acquire the VUI implementation of these SDKs by decompiling the official apk files using jadx [18]. These VUI implementations described by primitives are written in Roma's source code so it can automatically detect GV-race in Android apps that use these SDKs to implement their VUIs.

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

B. Evaluation Results

1) RQ1: Accuracy: The time limit to analyze each app is set to 30 minutes for all evaluation methods, including the time consumed by FlowDroid [20] to construct the call graph. The

TABLE II
THE DISTRIBUTION AND ACCURACY OF GV-race IN DIFFERENT DATASETS. **GV CO-ACCESS**: THE NUMBER OF APPS THAT CONTAIN RESOURCES ACCESSED BY BOTH GV ACTIONS.**GV-RACE**: THE NUMBER OF APPS WITH GV-RACE FOUND BY ROMA

Dataset	Type	GV co-access	GV-race	GV-race (confirmed)	$\frac{\text{GV-race}}{\text{total}}$	$\frac{\text{GV-race}}{\text{GV co-access}}$	accuracy
GV-in-app dataset	write-write	8	8	7	6.6%	100%	87.5%
	read-write	6	6	6	4.9%	100%	100%
	GV-race	10	10	9	8.2%	100%	90%
GV-between-apps dataset	write-write	36	36	31	40.9%	100%	86.1%
	read-write	20	20	18	22.7%	100%	90%
	GV-race	42	42	36	47.7%	100%	85.7%
system-level dataset	write-write	56	56	56	100%	100%	100%
	read-write	56	56	56	100%	100%	100%
	GV-race	56	56	56	100%	100%	100%

GV interaction of GV-in-app, GV-between-apps and system-level dataset is quite different due to different restrictions and behavior of the VUI. As a result, we record and analyze different levels of GV-race detected in different datasets respectively.

After the analysis result is acquired, we first compare the GV-race detection capabilities of the evaluated methods. Both GV-race instances and other data race instances are recorded. Because the baselines report races without providing counterexamples, we confirm GV-race occurrences by tracing back the decompiled APK files to identify the conflicting GV actions and writing scripts to reproduce the detected race between these events. For Roma, if the counterexample can be reproduced by scripts, GV-race is confirmed. Since Roma detects multiple types of GV-race across all datasets, we further break down its results according to the different GV-race types and datasets. The analysis and confirmation result is shown in Table II.

Summary of GV-race detected by Roma and the baselines in all datasets. Roma detects a total of 108 apps with GV-race, whereas the baseline tools fail to identify any GV-race instances. On the other hand, Roma detects only GV-race, while ER Catcher supports the detection of other types of data races and identifies 141 apps exhibiting GUI races, especially use-after-free races. Due to configuration issues, we cannot acquire nAdroid's testing results. The primary reason for the baseline methods' failure to detect GV-race is that the baseline methods are designed for GUI-level race detection and thus do not model fine-grained GV interactions or extract detailed VUI implementations. Specifically, nAdroid focuses on UAF races but is unable to detect the VUI-related UAF races captured by Roma. Compared with Roma, nAdroid does not track VUI-level callbacks and APIs, as well as their implementations. It focuses on capturing happen-before relationship between callbacks and lifecycle events, but lacks techniques to represent complex happen-before relationship aroused by traditional locks and happen-in-the-middle behavior. ER Catcher introduces the CSF to model the Android library, and while its functionality could theoretically be extended to support coarse-grained VUI implementation via domain-specific CSF functions, such a simplification prevents it from modeling complex GV interactions, such as happen-in-the-middle behavior. Moreover, ER Catcher does not model or analyze the happen-before relationships introduced by locks. Without precise modeling of GV interactions, it is difficult for ER Catcher to produce replicable counterexamples for GV-race.

In terms of detection scope, the baseline methods are general data race detectors in Android apps, but they may fail to capture VUI-related races. In contrast, Roma is a lightweight tool specifically designed to detect VUI-related data races. Overall, the detection scopes of Roma and the baseline methods are complementary. From the perspective of overhead, ER Catcher fails to analyze 31 apps within 30 minutes, while Roma fails on 2 apps. For the remaining apps, the ER Catcher requires an average of 134.3 seconds and 2,534.3 MB of memory per app, while Roma requires 129.8 seconds and 599.4 MB. The number of time-out apps for ER Catcher is 15.5 times that of Roma, and its memory consumption is nearly four times higher. The longer execution time and higher memory usage of ER Catcher are expected, as they capture all GUI events and performs global race analysis. In contrast, Roma focuses on VUI-related events and analyzes them in pairs.

GV-race inside tested apps found in the GV-in-app dataset. The first three rows in Table II show the number of apps with GV-in-app GV-race. Among all the GV-in-app apps, Roma records apps that contain conflicting GV actions (i.e., GV co-access apps) and finds 8 apps with write-write co-access and 6 apps with read-write co-access. Among 6 apps with GV read-write co-access, 2 of them are use-after-free races while the rest 4 apps contain normal VUI read and GUI write operations. Finally, Roma discovers that 6.6% (10/122) of GV-in-app apps with write-write GV-race and 4.9% of GV-in-app apps with read-write GV-race. Surprisingly, 100% of GV co-access apps exhibit GV-race, demonstrating the pervasiveness of this issue.

GV-in-app apps handle the VUI related resource themselves, enriching the categories of GV-race. For example, the VUI related resource is managed by the tested app, so this use-after-free GV-race could happen: the VUI related resource is used after being freed by a GUI action. Use-after-free is a specific type of read-write race, but it could lead to more severe consequences like crashes. Without careful design of the VUI behavior and the GV interaction, serious problems could occur.

GV-race between two apps found in the GV-between-apps dataset. The fourth to the sixth rows in Table II present the number of apps with GV-between-apps GV-race. Among all the GV-between-apps apps, Roma discovers 36 apps with write-write co-access and 20 apps with read-write co-access. In total, nearly a half of GV-between-apps apps are detected with GV-race.

Compared with the GV-in-app dataset, the GV-between-apps dataset exhibits more problems. One reason for the universality of GV-race is that GV co-access is common in GV-between-apps apps. We assume that their developers may be unfamiliar with the Android system or the race issues compared with developers that use the SDK to implement the VUI, so they may lack the awareness to prevent GV-race by avoiding GV co-access. Another reason is that the VUI implemented by another app cannot be destroyed or canceled by developers after it launches, so the prevention technologies for GV-race are restricted to the GUI level.

GV-race brought by system-level VUI actions found in the system-level dataset. The last three rows in Table II show the number of apps with system-level GV-race. All apps in the system-level dataset have GV co-access because system-level VUI actions have access to any widgets on the current activity. In addition, all these apps suffer from system-level GV-race.

The reason for the widespread existence of system-level GV-race is that system-level VUI actions cannot be interrupted by other apps, making it hard to prevent system-level GV-race from the app level. System-level apps do not call or implement the VUI themselves. They cannot refuse, interrupt or free the external VUI actions. As a result, their VUI behavior and GV interaction is fixed.

The accuracy of GV-race detection in three datasets. The last column of Table II shows the accuracy of GV-race detection. Through validation, we confirm 7 of 8 (87.5%) apps with write-write GV-race and 6 of 6 (100%) apps with read-write GV-race in the GV-in-app dataset. In total, 9 of 10 (90%) apps are confirmed with GV-race in the GV-in-app dataset. In the GV-between-apps dataset, we confirm 31 of 36 (86.1%) apps with write-write GV-race and 18 of 20 apps (90.0%) with read-write GV-race. In total, 36 of 42 (85.7%) apps are true positive in the GV-between-apps dataset. In the system-level dataset, we confirm 56 of 56 (100%) apps with both read-write and write-write GV-race. To sum up, Roma achieves an accuracy of 94.0% in detecting write-write GV-race and 97.6% in detecting read-write GV-race.

Due to the inaccuracy of entire call graphs obtained by off-the-shelf tools, there are false positives. The reason for false positives is mainly due to variable polymorphisms and function polymorphisms. It is difficult to accurately obtain the specific callees of all function calls, so the callees of call relationships may be wrong in the call graph. As a result, our analysis based on these call relationships is inaccurate on specific cases. Secondly, functions in the call graph may be unreachable. For example, a widget related to the launch of the VUI is always invisible if the value of a boolean variable v is constantly true. In this case, an on-demand data-flow analysis on the variable v is required to prove that the VUI does exist. However, mitigating this issue is complex and time-consuming but cannot improve the accuracy evidently.

Feedback from developers on reported issues. We contact the developers via emails. Among 45 confirmed apps, we are able to find the email addresses of developers for 32 apps and send issue reports accordingly. Ultimately, we receive feedback from the developers of four apps. They either discuss approaches to using the app to avoid the race or state that they have received our feedback and reported to their technicians.

The four apps cover a range of functionalities, including dictionary, app management, and notebook. Two of them (20.0%) are from the GV-in-app dataset, while the remaining two (4.8%) are from the GV-between-apps dataset. We observe that developers of GV-in-app apps are more willing to address GV-race, likely because they implement the VUI themselves, giving them deeper knowledge of GV interactions and a greater sense of responsibility for handling GV-race. Across these apps, we identify 43 usability issues and 1 runtime error. Analysis of the usability issues indicates that problems within the same app tend to follow similar patterns, as they are mostly associated with a single VUI action. For example, in the GV-between-apps app "Persian Dictionary - Dict Box", conflicting GUI actions are blocked by a dialog during execution of the core VUI action. However, if conflicting GUI actions are triggered after the VUI action begins but before it completes, multiple instances of GV-race can occur. The single runtime error is caused by a use-after-free instance of GV-race, occurring because the VUI handler is freed by "onStop" before being used by "onResults". This error has been included as a case study in Section VI-B.

Although a large number of GV co-access apps in our dataset are identified with GV-race, it does not mean that GV-race definitely occurs in GV co-access apps. Actually, there are strategies taken to prevent GV-race but they fail to completely avoid it. We conduct the case study to analyze developers' implementation of GV interaction in Section VI-B. After a detailed analysis of GV-race patterns, we find that GV-race can be avoided by adding locks at specific locations, which is recommended in Section VI.

> **Answer to RQ1**: Compared with GUI-level race detectors, Romadetects more GV-racebenefited from its detailed VUI implementation extraction and accurate GV interactions modeling. Romafinds that GV-race broadly exists in apps that contain resources accessed by both VUI actions and GUI actions in GV-in-app dataset, GV-between-apps dataset and system-level dataset, which suggests that developers are not aware of the effective strategies to prevent GV-race. The accuracy of Roma to detect GV-raceis high, with a true positive rate of 94.0% and 97.6% on write-write GV-raceand read-write GV-race, respectively.

2) RQ2: Efficiency: To measure the efficiency of Roma, we record the analysis time of apps in our dataset. We classify the apps into three categories based on different analysis time. (1). Efficient apps are inliers in the box plot. (2). Outlier apps can

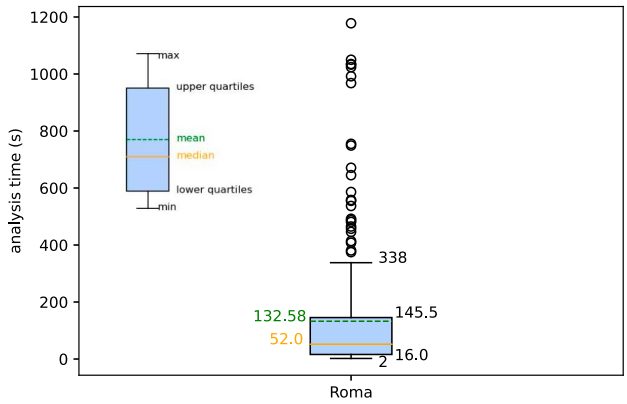


Fig. 7. The efficiency of Roma.

be analyzed in 30 minutes but are outliers in the box plot. (3). Time-out apps cannot be analyzed in 30 minutes.

Distribution of analysis time. Fig. 7 shows the distribution of analysis time required by Roma on all datasets, excluding 2 time-out apps. The vertical axis represents the analysis time measured in seconds. The green dotted line marks the mean value, while the orange line represents the average value. As the figure shows, both the average and mean analysis time is less than 150 seconds. In total, 95.1% (253/266) of the apps can be analyzed in 10 minutes. Only 0.75% (2/266) of the apps are time-out apps.

Roma's high efficiency benefits from the pairwise GV analysis framework. A pair of GV actions that may access the same resource is taken as the minimum detection unit. This strategy helps reduce the scale of the GV interaction graph and the solver's checking time. In addition, an app could contain multiple GV pairs with similar behavior, and Roma reuses the previous results to avoid repeatedly checking them.

Although most apps can be analyzed in time, we also find 25 outlier apps in Fig. 7 and 2 time-out apps. We analyze these apps and discover two factors that influence the efficiency. The time required by FlowDroid to construct the entire call graph accounts for a large proportion of the total analysis time. What's more, due to variable polymorphisms and function polymorphisms, FlowDroid cannot always accurately find callees. We explore as many candidate callees as possible to avoid missing races, but such a strategy can influence the call graph exploration time. These apps only account for a small proportion.

Relationship between apps' size and analysis time. In order to evaluate Roma's scalability on large apps, we use two metrics to measure Roma's efficiency on different sizes of apps. The first metric is the percentage of efficient apps. It is calculated by dividing the number of efficient apps with the number of all apps whose size is within a given range. The second metric is the average time. It is the average analysis time of efficient apps and outlier apps because the analysis time of time-out apps is not recorded.

Table III shows the percentage of efficient apps and average analysis time of apps with different sizes. The percentage of efficient apps slowly decreases with the increase of apps' size, but it remains higher than 81%. The average analysis time slowly increases with the growth of apps' size, but it remains lower than 200 seconds even for large apps. As the number of apps whose sizes exceed 70MB is smaller than the other categories in our dataset, the data reported in the last column may not be representative.

TABLE III
THE PERCENTAGE OF EFFICIENT APPS AND AVERAGE ANALYSIS TIME FOR APPS OF DIFFERENT SIZES. THE EFFICIENT APPS ARE THOSE INLIERS IN FIG. 7

size of apps (MB)	≤ 5	(5, 10]	(10, 30]	(30, 70]	>70
efficient apps (%)	100.0%	93.3%	89.5%	81.1%	92.5%
average time (s)	36.3	92.1	162.7	182.5	102.1

The results demonstrate that the impact of apps' size on Roma's efficiency is limited. The pairwise detection strategy ensures that Roma is hardly affected by the apps' scale. No matter how large the app is, the minimum detection unit is a pair of GV actions, and results are reused for similar patterns. Therefore, the analysis time grows linearly with the number of different GV-race patterns, rather than exponentially with the increase in apps' scale.

Effectiveness of the pairwise analysis strategy in improving efficiency. To evaluate the effectiveness of our pairwise strategy, we implement Roma-$Pairwise$ and compare its analysis time with that of Roma. Specifically, Roma-$Pairwise$ first extracts the GV-related call graph for all VUI events and potentially conflicting GUI events. The GV interaction graph is then constructed for this comprehensive call graph using Algorithm 1. Finally, constraints are encoded and checked based on the GV interaction graph. Because the pairwise strategy is designed to reduce model-checking time, and apps without GV co-access are not checked, this experiment is conducted only on GV co-access apps. According to RQ1, there are 108 GV co-access apps.

Fig. 8 presents a candlestick chart comparing the analysis time required by Roma and Roma-$Pairwise$ on GV co-access apps. On average, Roma requires 182.2 seconds, whereas Roma-$Pairwise$ requires approximately 1.46 times this duration. We also observe that Roma-$Pairwise$ fails to complete the analysis of 2 GV co-access apps within the allotted time. The results indicate that the effectiveness of the pairwise detection strategy increases with the complexity of GV interactions. For a VUI event with multiple conflicting GUI events, the GV interaction patterns for each pair are often similar. Using the pairwise strategy, it suffices to check a single pair of GV events and reuse the results for other similar pairs. In contrast, global analysis may examine GV interactions involving potentially hundreds of events simultaneously. These findings demonstrate that the pairwise strategy significantly improves analysis efficiency.

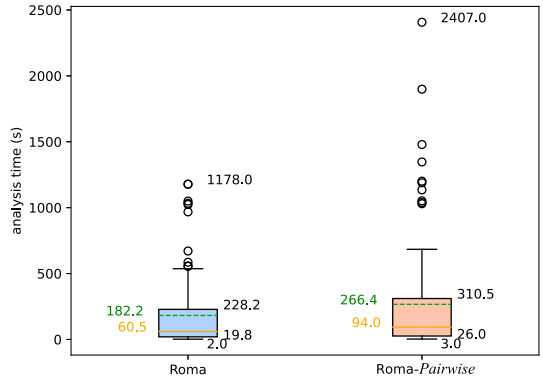


Fig. 8. The distribution of analysis time required by Roma and Roma-$Pairwise$ to analyze the GV co-access apps.

Answer to RQ2: In general, Romacan analyze over 95% of apps in 10 minutes. As a comparison, only 0.75% of apps cannot be analyzed in 30 minutes. With the growth of apps' sizes, the average analysis time increases slowly, but remains lower than 200 seconds. The results demonstrate that Romahas an efficient testing performance, regardless of apps' size. We also implement Roma-$Pairwise$ and compare its analysis time with Romato evaluate the effectiveness of our pairwise strategy. The results demonstrate that Romareduces the average analysis time by approximately 46%.

3) RQ3: Ablation Study: To assess the effectiveness of our formalization part in accurately detecting GV-race, we conduct an ablation study. Since Roma is an end-to-end framework, it is infeasible to remove the entire formalization module. To address this issue, we preserve the primitives and rules commonly employed in GUI-level data race detection for Android apps and apply the standard data race constraint for GV-race detection. The GV-related call graph is constructed following the same approach as Roma. Specifically, GV-related primitives, including disable, enable, speechRecognition and VUIsink, as well as the associated Rule 9 are removed. VUI sinks are mapped to the primitive GUIsink. The GV-race constraint is replaced with the standard data race constraint: postEvent $(\_, click_{GUI2}, t_2) \leq$ GUIsink1 $(t_1) \wedge$ postEvent $(\_, clickGUI1, t_1) \leq$ GUIsink$_2$ (t_2). We denote this modified implementation as Roma-$Forml$ and compare it with the original Roma to demonstrate the effectiveness of our design.

Table IV presents the number of apps with GV-race, the number of confirmed cases, and the corresponding accuracy rates detected by Roma and Roma-$Forml$. We adopt the same validation approach as in RQ1 to confirm GV-race, that is writing a script to replicate the counterexample. If the counterexample cannot be reproduced, it is classified as a false positive. The results show that Roma-$Forml$ consistently achieves lower accuracy across all types of GV-race and datasets. In particular, on the GV-between-apps dataset, Roma-$Forml$ attains the lowest accuracy of 0% for all categories of GV-race. Compared with Roma, its detection accuracy for GV-between-apps GV-race is the poorest. The second-worst performance is observed in detecting GV-in-app GV-race, where the accuracy drops to about 60%, approximately 30 percentage points lower than that of Roma. The best performance of Roma-$Forml$ is in detecting system-level GV-race, where it achieves an accuracy of 100%.

The low accuracy of Roma-$Forml$ in detecting GV-between-apps GV-race primarily stems from its inability to capture happen-in-the-middle behaviors. In these apps, the VUI invoked by the apps typically triggers a dialog at launch, which may disable conflicting GUI actions. Without the fine-grained primitives and rules introduced in Roma, Roma-$Forml$ fails to model these constraints accurately, thereby causing false positives. For apps in the GV-in-app dataset, the VUI implementations are generally more diverse. Some involve happen-in-the-middle behaviors, which are inherently difficult to model and analyze; others adopt simpler designs, which can be represented accurately with existing primitives and rules. Another source of inaccuracy lies in the general data race constraints, which include GUI-before-VUI GV-race. In practice, this type of GV-race rarely occurs, since GUI actions typically respond much faster than VUI actions. Apps in the system-level dataset achieve the highest accuracy. Their VUI actions are executed at the system level and cannot be interrupted or disabled by app-level GUI actions. As a result, the GV interaction graphs for these apps are straightforward to construct and exhibit consistent patterns across apps. In summary, Roma-$Forml$ achieves higher accuracy for apps with simple GV interactions but struggles to model and analyze those with complex interaction behaviors.

We also measure the time and memory consumption of Roma-$Forml$. Both Roma-$Forml$ and Roma have 2 timeout apps. For the remaining apps, Roma-$Forml$ and Roma require an average of 130.6 and 132.6 seconds, respectively, and consume an average of 664.1 MB and 651.1 MB of memory. Notably, Roma can not analyze 13 apps within a 1 GB memory limit, while Roma-$Forml$ fails on 9 apps under the same constraint. Compared with Roma, Roma-$Forml$ costs fewer time and memory usage, although the advantage is marginal. Despite considering fewer primitives and rules, Roma-$Forml$ still performs the full process of call graph extraction, interaction graph construction, and constraint solving. Given its lower accuracy in detecting GV-race, this computational cost is considered acceptable.

Answer to RQ3: We remove the VUI-related primitives, rules, and constraints to implement Roma-$Forml$, and compare it against Romato evaluate the contribution of our formalization. The results show that Roma-$Forml$ struggles to accurately detect all types of GV-race, particularly GV-in-app and GV-between-apps apps. The GUI-level primitives, rules, and constraints are sufficient only for apps with simple GV interactions. These findings confirm that our formalization substantially enhances the performance of GV-race detection.

TABLE IV
THE DISTRIBUTION AND ACCURACY OF GV-RACE DETECTED BY ROMA AND ROMA-*FORML* IN DIFFERENT DATASETS

Dataset	Type	Roma			Roma-$Forml$		
		GV-race	GV-race(confirmed)	accuracy	GV-race	GV-race(confirmed)	accuracy
GV-in-app dataset	write-write	8	7	87.5%	8	5	62.5%
	read-write	6	6	100%	6	4	66.7%
	GV-race	10	9	90%	10	6	60.0%
GV-between-apps dataset	write-write	36	31	86.1%	36	0	0.0%
	read-write	20	18	90%	20	0	0.0%
	GV-race	42	36	85.7%	42	0	0.0%
system-level dataset	write-write	56	56	100%	56	56	100.0%
	read-write	56	56	100%	56	56	100.0%
	GV-race	56	56	100%	56	56	100.0%

4) RQ4: Applicability: The GV-in-app dataset only contains apps whose VUI is implemented by the Google SDK because the Google SDK is the most popular one. To further evaluate the applicability of Roma's framework to apps whose VUIs are implemented by using the other SDKs, we conduct this experiment. Among the top-10 popular speech recognition service providers as introduced in Section II-A, six of them provide Android versions to implement the VUI, along with detailed documentation and sample apps. Except from the Google SDK, the rest five SDKs are from Alibaba, Baidu, Huawei, Microsoft and Tencent.

We manage to collect a dataset by scanning the inclusion of the other five VUI SDKs' APIs in the decompiled apk files from the initial large dataset. However, we find that third-party developers merely use the other five SDKs to implement the VUI. There might be financial and privacy concerns to use the other five SDKs. Firstly, Google SDK allows developers to use phone-embedded speech recognition service while other five SDKs only provide APIs to access their paid speech recognition service. In addition, using the paid speech recognition service requires authentication by using secret keys. If the secret key is provided by the developer and embedded in the source code, there might be a high risk of privacy leakage. On the other hand, the other five SDKs are mostly used by the official app developers. To further enrich our dataset, we collect official apps developed by the same developers as those SDKs and scan weather the related APIs are included. We then conduct a further filtration using the same three rules as collecting the GV-in-app dataset. Finally, we get 5 apps and we label it as the GV-in-app (third-party) dataset.

After running Roma on the GV-in-app (third-party) dataset, we find that there are no GV co-access apps, which means these apps do not suffer from GV-race. We validate the results by manually running the apps and reading the decompiled apk files. However, GV-race can happen if the other five VUI SDKs are carelessly used, and Roma is able to detect GV-race.

To prove this, we download sample apps of these five VUI SDKs and mutate them to form the GV-in-app (self-developed third-party) dataset. We implement two mutations according to patterns used by developers that use the Google VUI SDK. The first mutated version uses the main thread as the VUI thread. To implement this, we use the main thread, i.e., the thread to handle the UI events, to call the API that launches the VUI. It is labeled as the "main as VUI" version. The second mutated version forks a new thread as the VUI thread and uses the VUI thread to call the API that launches the VUI. It is labeled as the "new VUI thread" version. These mutated apps can be found on our website [15].

TABLE V
NUMBER OF MUTATED SAMPLE APPS WITH/WITHOUT GV-RACE

With GV-race		With GV-race	
main as VUI	new VUI thread	main as VUI	new VUI thread
1	0	4	5

The number of mutated sample apps with GV-race detected by Roma is displayed in Table V. Among them, Roma finds 9 mutated sample apps with GV-race, including 4 "main as VUI" apps and 5 "new VUI thread" apps. There are 100% of "new VUI thread" apps with GV-race, and 80% of "main as VUI" apps with GV-race.

The only app without GV-race is the app that uses the main thread as the VUI thread and adopts the synchronous VUI implementation. As the VUI thread (also the main thread in this case) in the synchronous VUI implementation waits for the ASR results to return, the main thread could be blocked for over five seconds. However, blocking the main thread leads to crashes. Following the validation approach in RQ2, we manually write scripts to reproduce the counterexample given by Roma on these apps. Our manual validation proves that the testing results is 100% accurate.

Roma's applicability across different SDKs benefit from its primitive-level interaction abstraction and problem detection. Despite different SDKs, the VUI implementation follows similar patterns as shown in Fig. 1. After abstraction, the GV

interaction patterns can be presented by our pre-defined primitives and tied by happen-before rules. To test apps implemented by unknown VUI SDKs, testers only need to add new mapping relationships from VUI APIs to our pre-defined primitives. The upcoming process is at the primitive level, which can be automatically done by Roma.

Answer to RQ4: We implement mutations on sample apps of five VUI SDKs and get 10 news apps to form a new dataset. There is only one app without GV-race, but it blocks the main thread and leads to crashes. All 5 apps that fork a new thread as the VUI thread and 4 apps that use the main thread as the VUI thread are detected and confirmed with GV-race. The results prove that Roma is suited for analyzing apps implemented by other VUI SDKs.

VI. Discussion

A. Consequence of GV-Race

In our experiments, we observe that GV-race occurs in a variety of apps. By validating GV-race using replication scripts, we identify and summarize two main types of consequences resulting from GV-race.

- **Usability issues.** GV-race is often difficult for typical users to detect and correct, particularly because VUIs are designed to assist the elderly and individuals with disabilities. During the execution of a VUI action, users may inadvertently trigger conflicting GUI actions, yet struggle to recognize that the results deviate from expectations. When using the VUI requires additional effort, it undermines the original goal of improving convenience and accessibility. Analysis of real-world apps with GV-race shows that GV-race occurs across various categories, including navigation, translation, food delivery, and note-taking apps. In Section VI-B, we present two case studies on translation and note-taking apps to illustrate these issues. Our analysis of real-world apps reveals that 101 (100%) GV-in-app and GV-between-apps apps exhibiting GV-race experience usability issues.
- **Errors.** GV-race can also result in explicit errors, particularly when use-after-free races occur. In such cases, the app attempts to access a resource that has been set to `NULL` or released, which can lead to error messages or crashes. This type of GV-race can significantly degrade user experience and application reliability. In our experiments, we find that 1.96% of apps exhibiting GV-race experience errors.

We observe that most instances of GV-race are usability issues. Explicit runtime errors are primarily caused by use-after-free races [6], [8]. However, Roma detects use-after-free races only within GV interactions. Moreover, read and write operations are common in GV interactions, while free operations are relatively rare, especially in apps that interact with VUIs only through intents or system events. This is because internal GUI actions typically lack permission to free resources maintained by external VUIs, and external VUI actions generally perform read and write operations on visible widgets rather than freeing resources. In our experiments, Roma detects runtime errors only in the GV-in-app dataset. Specifically, it identifies 9 apps with 48 distinct instances of GV-race, including 2 apps with 2 runtime errors and all 9 apps with 46 usability issues. In the GV-between-apps and system-level datasets, Roma detects 36 apps with 114 instances of GV-race and 56 apps with 22,130 instances of GV-race, respectively. For system-level apps, nearly all GUI sources and system-level VUI events exhibit GV-race, all of which are usability issues.

These two types of consequences are derived from the results of our experiments. Additionally, GV-race may pose security risks if exploited in financial or smart home apps. Consequently, it is essential for developers to pay attention to and mitigate GV-race.

B. Case Studies and Root Cause Analysis

In RQ1 (see Section V-B1), we surprisingly discover that a large number of GV co-access apps suffer from GV-race. In RQ2, we confirm that 45 apps have GV-in-app GV-race or GV-between-apps GV-race. To analyze the root cause of GV-race, we decompile the APK files of all 45 apps, identify the callbacks associated with conflicting GV actions based on Roma's report, and examine their implementations. We focus particularly on locks and APIs that change the layout or directly disable and interrupt GV actions. This entire process requires less than one day to complete. Through this analysis, we find that these apps' developers do not take proper strategies to prevent GV-race. To further study the issue, we conduct case studies on two apps; one is an app with GV-in-app GV-race, and another is an app with GV-between-apps GV-race.

Case study on an app with GV-in-app GV-race. The app found with GV-in-app GV-race is called the "Voice Input Notebook" app. Fig. 9 shows a simple graph of a conflicting GV action pair in this app. It calls the Google VUI SDK to implement its VUI, which belongs to the asynchronous VUI implementation. So the main thread (which is also the VUI thread) is idle and *unprotected* (marked in blue) during the ASR process, making it possible to insert a conflicting GUI action. The developer does not take extra strategies like disabling the conflicting GUI actions, or interrupting the VUI process when the conflicting GUI action is invoked to protect the idle main thread.

This app also exhibits a use-after-free race. Fig. 10 illustrates a simplified graph of a GUI free and VUI use action pair. After the user stops the activity, the $click_{GUI}$ event is posted to the main thread. However, before it is executed, the ASR result returns and posts the $event_{ASR}$ event to the main thread. Then, $click_{GUI}$ and $event_{ASR}$ are executed sequentially. Although the cancel function invoked during the execution of $click_{GUI}$ prevents $event_{ASR}$ from being posted, it cannot remove $event_{ASR}$ if it has already been appended to the queue. In this scenario, the subsequent use of the VUI handler in $event_{ASR}$ after it has been freed in $click_{GUI}$ triggers a NullPointerException error.

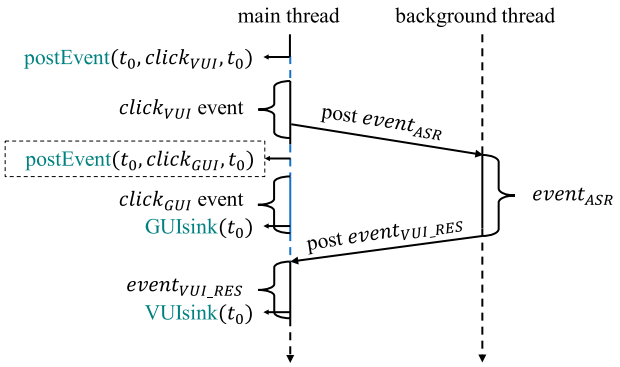


Fig. 9. Simple graph of the GV interaction patterns in an app with GV-in-app GV-race.

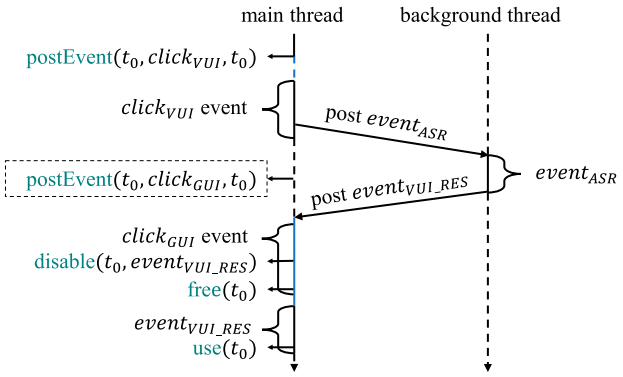


Fig. 10. Simple graph of the GV interaction patterns in an app with GV-in-app use-after-free GV-race.

Roma accurately models the VUI framework, including posting the $event_{ASR}$, conducting the $event_{ASR}$ in the background thread and returning ASR results to the main thread by posting the $event_{VUI\_RES}$ task, making it possible to identify that there is an idle period in the main thread when the background thread is running the ASR task. Furthermore, the behavior of the cancel function is accurately modeled by the disable primitive, which enables Roma to identify that $event_{ASR}$ must be posted after the post of $click_{GUI}$ and before the execution of $click_{GUI}$ in order to trigger GV-race.

Case study on an app with GV-between-apps GV-race. The app with GV-between-apps GV-race is called "Burmese English Translator". This app calls an intent to start the Google Speech Recognition service provided by the "Speech Recognition & Synthesis" app to conduct the VUI action. Different from the Google VUI SDK, the VUI implemented by another app is more encapsulated and implements strategies to protect the main thread. Specifically, a dialog opens when the VUI launches and closes after ASR results return. The started dialog could disable conflicting GUI actions, making it harder for GV-race to happen. Unfortunately, this strategy cannot completely avoid the race.

Fig. 11 is a simple graph to illustrate the GV interaction in this app. In the $click_{VUI}$ event, a dialog opens and disables the conflicting GUI action (disable $(t_0, click_{GUI})$). The conflicting

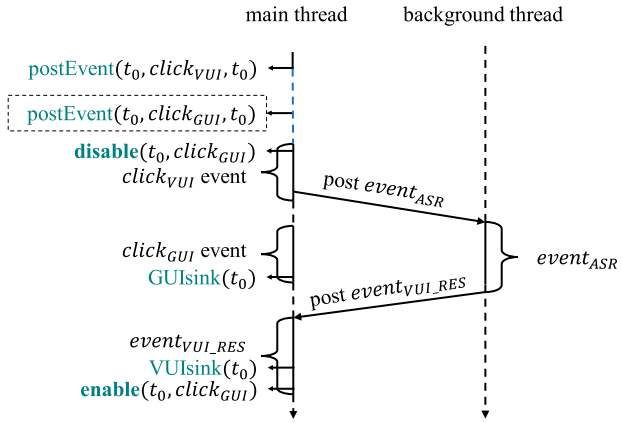


Fig. 11. Simple graph of the GV interaction patterns in an app with GV-between-apps GV-race.

GUI action is enabled after the VUI sink. Although the post of the conflicting GUI action postEvent $(t_0, click_{GUI}, t_0)$ cannot happen from the start of the $click_{VUI}$ event to the end of the $event_{VUI\_RES}$ event, it can be invoked after the post of the VUI action postEvent $(t_0, click_{VUI}, t_0)$ and before the start of the $click_{VUI}$ event. Once the $click_{GUI}$ event is posted to the event queue in the main thread, the main thread will execute it when it is idle. In this case, the main thread executes it when the background thread is running the ASR task, so the GUI sink returns before the VUI sink.

Roma can model complex GV interaction using specific primitives like disable and enable. By adding happen-before constraints based on Rule 9, Roma identifies that the $click_{GUI}$ event cannot be posted to the main thread when the background thread is running the $event_{ASR}$. As a result, there is only one idle period (marked in blue) for users to post the $click_{GUI}$ event and trigger GV-race.

To sum up, the root cause of GV-race is that the main thread is unprotected from the unrestricted conflicting GUI action when the ASR process runs.

C. Recommended Strategies to Prevent GV-Race

We discover that GV-race broadly exists in apps and developers are not aware of GV-race and its effective prevention strategies (see Section VI-B). As a result, it is important to recommend effective strategies to prevent GV-race. We propose mitigation strategies for both developers and the system to address GV-race.

Mitigation strategy for developers. GV-in-app and GV-between-apps GV-race are recommended to be mitigated by app developers. Preventing GV-race at the SDK level is challenging because the SDK only provides the essential components for VUI functionality, including APIs for performing ASR tasks and callbacks for handling ASR results. The SDK does not impose restrictions on VUI sinks; therefore, it is the responsibility of developers to manage the access sequence of GV sinks. Since we cannot forbid users from invoking two potentially conflicting GV actions, we suggest to arrange their responses

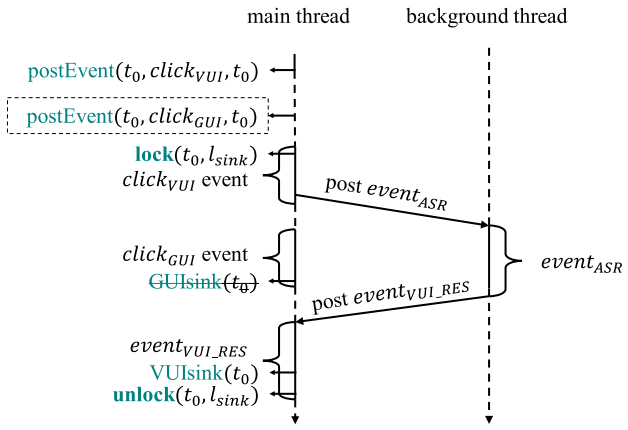


Fig. 12. Recommended approach: add locks.

in sequence. To achieve that goal, we recommend adding locks to ensure the atomicity of the VUI action.

Fig. 12 illustrates the approach to adding locks. The lock of GV-shared resource is acquired in the $click_{VUI}$ event and released in the $event_{VUI\_RES}$ event. In this case, the GUIsink (t_0) cannot happen before the VUIsink (t_0) as the GV-shared resource is locked when the main thread is idle. Meanwhile, the main thread is not blocked, so it can respond to other UI actions during the ASR process. In detail, there are two implementations of the $click_{GUI}$ event. In a simple version, the $click_{GUI}$ event tries to acquire the lock before the GUIsink (t_0). If it fails to acquire the lock of the GV-shared resource, GUIsink (t_0) will not be executed before the $event_{VUI\_RES}$ event ends. In this implementation, the GUI action may not be responded if it has conflicts with another VUI action. In another implementation, we recommend delaying the GUIsink (t_0) when the lock cannot be acquired. Specifically, the $click_{GUI}$ event uses the postDelayed function to post another $event_{GUIsink}$ event to the main thread and delays a few seconds for execution. The $event_{GUIsink}$ event tries to acquire the lock again and executes the GUIsink (t_0). This process will continue until the lock is successfully acquired and the GUIsink (t_0) is executed. This implementation ensures that GUIsink (t_0) is executed after VUIsink (t_0).

To demonstrate the effectiveness of our mitigation strategy, we implement it on the mutated apps used in RQ3. Since the source code of the real-world apps confirmed to exhibit GV-race is unavailable, we can not apply the mitigation strategy to these apps. In addition to the nine mutated apps previously detected with GV-race, we implement two additional apps following the "main as VUI" and "new VUI thread" patterns using the Google VUI SDK. These two apps are also confirmed to exhibit GV-race. The mitigation strategy is applied to all eleven apps, and subsequent analysis with Roma confirms that they are free from GV-race. Running the enhanced apps further validates the strategy: conflicting GUI actions can still be invoked during the ASR process, but the conflicting GUI sinks do not execute, and other unrelated GUI actions are unaffected.

Mitigation strategy at the system level. System-level GV-race can hardly be mitigated by app developers, as apps cannot observe or interrupt system-level VUI actions. These system-level VUI actions can access any widgets within the current activity when the user issues the appropriate voice command. Consequently, system-level GV-race arises on GUI-action-accessible widgets when system-level VUI actions are active. We propose two mitigation strategies at the system level to mitigate system-level GV-race.

The first strategy is to disable all GUI actions on the current activity once system-level VUI actions are enabled. However, this approach may reduce efficiency due to the relatively long response time of VUI actions. The second strategy is to interrupt system-level VUI actions whenever a GUI action is invoked. This approach assumes that GUI actions always take priority over VUI actions.

D. Limitation and Threads to Validity

Limitations. The limitations of Roma come from two parts. Firstly, the entire call graph is built by third-party tools like FlowDroid. Due to the variable polymorphisms and function polymorphisms, the call graph may contain call relationships with unreachable or inaccurate callees. This can affect Roma's analysis accuracy. Furthermore, Roma is a framework specially designed for detecting data races associated with GV interactions, and thus it may not be capable of identifying data races unrelated to VUI or other forms of VUI-related concurrency issues. Nevertheless, by leveraging our source code–level extraction of VUI behaviors and the formalization of GV interactions at the primitive level, we anticipate extending our approach to analyze additional VUI-related concurrency problems in future work.

Threats to Validity. The first potential threat to validity lies in the selection of apps. These apps are downloaded from Apkpure [19]. They may not be representative globally. However, we try to include popular apps by crawling a large dataset that contains over 10,000 top-popular apps covering all categories and form our GV-in-app, GV-in-app (third-party), GV-between-apps and system-level datasets from it. Since GV-in-app, GV-in-app (third-party) and GV-between-apps datasets require apps that adopt specific VUI implementations but a large number of the 10,000 apps do not support the VUI or use self-defined libraries to implement the VUI, we gather these datasets by scanning the 10,000 apps to find qualified apps. Considering the other VUI SDKs are not commonly used by third-party app developers due to financial and privacy concerns, we enrich the GV-in-app (third-party) dataset by including official apps developed by the same VUI SDK developer, along with the mutated sample apps.

The second threat to validity arises from the design of primitives. Our primitives are designed to model the GV interaction patterns and are summarized from existing GV interaction designs. As a result, the set may not be exhaustive, particularly if new GV interaction mechanisms are introduced. Nevertheless, our current primitive and rule set capture a wider range of constraints and achieve greater accuracy in detecting GV-race

compared with related race detection approaches. We plan to further refine and extend these primitives and rules to enhance their coverage and reliability in future work.

The third threat arises from the mapping of APIs to primitives. We establish mappings from APIs in the Android SDK and mainstream VUI SDKs to primitives for constructing the GV interaction graph. However, if the APIs undergo significant changes, such as modifications to their signatures or behaviors, this mapping information must be manually updated to ensure the continued correctness of Roma.

There is also threat related to baselines' results in the evaluation. We fail to acquire the detection results of baselines in some cases. To ensure the correctness of our running environment, we use baselines to analyze apps in their original datasets and get expected results. We also try to contact the developers for help. From our perspective, these tools have the ability to detect non-VUI related data races in Android apps.

Another threat is that we only use FlowDroid [20] to construct the call graph. We believe that this tool is representative, as it is a well-known and constantly updated call graph construction tool in Android apps.

E. Potential Solutions to Limitations for Future Work

Our analysis indicates that apps affected by GV-race predominantly exhibit usability issues. Although we do not observe real-world cases with severe consequences, such consequences may arise depending on the application scenario. For instance, scenarios involving payments require high result accuracy; otherwise, usability issues may lead to financial losses. With the continued development of multimodal technologies, we believe that GV-race will manifest in diverse forms. As future work, we plan to systematically investigate the symptoms and real-world impacts of GV-race in mobile apps.

Our dataset consists of apps whose VUIs are implemented using off-the-shelf SDKs or intents. However, we observe that some apps employ self-defined VUI implementations. Due to the inconsistent naming conventions and implementation patterns of such VUIs, it is difficult to identify, model or analyze them. To extend the applicability of Roma, we aim to explore the use of large language models to automatically predict VUI sources and map the corresponding APIs to our defined primitives in the future.

Compared with existing race detection tools for Android apps, Roma is a lightweight tool specifically designed to detect GV-race. As future work, we aim to develop a more comprehensive data race detection framework that supports not only data races triggered by VUI inputs or purely GUI-based inputs, but also those arising from other modalities.

VII. Related Work

Race detection on mobile apps. Dynamic program analysis is commonly used for race detection. Hsiao et al. [6] presented CAFA to find use-after-free violations between high-level operations in the Android system. They firstly introduced the causality model to represent the behavior of the event-driven program. DroidRacer [13] is a dynamic race detection technique that formalizes the concurrent semantics of the Android programming model. They defined happen-before relations of common execution sequences in the event-driven android system. AATT+ [21] that extended AATT [22] works by exploring the app to find potentially conflicting resource accesses, and then pressure-testing them to detect concurrency bugs. More works [23], [24], [25], [26] conducted automated GUI testing by exploring the app to cover more code and detect problems. However, above-mentioned dynamic analysis techniques may not build a complete system model without running the app thoroughly as the model is built by gathering execution traces sent by pre-instrumented VM or android libraries. As a result, dynamic program analysis techniques could miss many bugs.

In recent years, static analysis has become a popular approach for detecting race events. DeVA [27] detects "event anomalies" where read-write or write-write operations access the same resources. SIERRA [11] automatically generates order among lifecycles and GUI events using a harness-based model, and uses symbolic analysis to order the actions and memory accesses. ER Catcher [12] 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.

These race detection techniques are aimed at detecting GUI-related races on apps. They have limitations in detecting VUI related races as VUI related behavior cannot be represented without GV-race related primitives and happen-before relations. Compared with these tools, Roma is a pure static analysis tool specially designed for GV-race detection. Roma introduces GV-race-related primitives to model the VUI behavior and GV interaction. GUI exploration is not required, so that false negatives aroused by this process are avoided and testing efficiency is improved. Roma takes a pair of potentially conflicting GV actions as the minimum detection unit, reducing the scale of the model (i.e., Roma interaction) and analysis time. The detailed comparison between Roma and related tools is shown in Table VI.

Race conditions in traditional areas. Several researches defined [28] and classified [29] race conditions. Farah et al. [30] studied the impact of data races on UNIX systems. Researches [31], [32] detected the data race when two processes access the same file in the Unix environment. Other research [33] managed to detect and prevent the data race in file systems. Flanagan et al. [34] presented rccjava to detect race conditions in small to medium-scaled java programs and later extended it for large, realistic programs [35]. RacerX [36] uses flow-sensitive, interprocedural analysis to detect race conditions in large, complex multi-threaded systems. Licker et al. [37] 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 [38] and EventRacer [39] adapt the happen-before analysis to web and event-based applications to detect races.

TABLE VI
THE COMPARISON BETWEEN ROMAAND RELATED ANDROID RACE DETECTION TOOLS. RACE TYPE CONTAINS THREE KINDS OF LABELS: U REPRESENTS USE-AFTER-FREE RACE, R REPRESENTS READ-WRITE RACE (EXCLUDING USE-AFTER-FREE RACE), W REPRESENTS WRITE-WRITE RACE

	Fine-Grained Model of GV Interaction	Detailed Extraction of VUI Implementation	Detection Strategy	Race Type	Systematicness	Availability
CAFA [6]	×	×	global	u	app-level	×
DroidRacer [13]	×	×	global	r+w+u	app-level	×
EventRacer-Android [7]	×	×	global	r+w	app-level	× (broken)
SIERRA [11]	×	×	global	r+w+u	app-level	× (broken)
nAdroid [8]	×	×	global	u	app-level	✓
ER Catcher [12]	×	×	global	r+w+u	app-level	✓
Roma	✓	✓	pairwise	r+w+u	app- and system-level	✓

VIII. CONCLUSION

In this paper, we propose, formalize and define GV-race in Android apps. GV-race happens when the response sequence of the GV actions is disrupted. To detect GV-race, we propose the Roma framework. Roma firstly extracts the GV related call graphs. Based on these graphs, it builds the GV interaction graph using GV interaction related primitives. Finally, it encodes happen-before constraints on the GV interaction graph and solves them to detect GV-race. Roma analyzes 266 Android apps with both the GUI and the VUI and finds 101 apps with validated write-write, read-write or use-after-free GV-race. Our work reveals that careless usage of VUI SDKs can cause GV-race and developers are not aware of preventing GV-race.

ACKNOWLEDGMENT

We are grateful for the constructive feedback of all the anonymous reviewers to improve this manuscript.

REFERENCES

[1] "Mobile accessibility at W3c," 2018. [Online]. Available: https://www.w3.org/WAI/standards-guidelines/mobile/
[2] J. Díaz-Bossini and L. Moreno, "Accessibility to mobile interfaces for older people," in *Proc. 5th Int. Conf. Softw. Develop. Enhancing Accessibility Fighting Info-Exclusion (DSAI)*, vol. 27. Elsevier, 2013, pp. 57–66, doi: 10.1016/j.procs.2014.02.008.
[3] "Statista - the statistics portal for market data, market research and market studies," 2007. [Online]. Available: https://www.statista.com
[4] "Frequency with which smartphone owners use voice-enabled technology worldwide in 2017," 2017. [Online]. Available: https://www.statista.com/statistics/787382/worldwide-voice-technology-utilization/
[5] "Android.speech—Android developers," 2009. [Online]. Available: https://developer.android.google.cn/reference/android/speech/package-summary
[6] C. Hsiao et al., "Race detection for event-driven mobile applications," in *ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, Edinburgh, United Kingdom, 2014, New York, NY, USA: ACM, 2014, pp. 326–336, doi: 10.1145/2594291.2594330.
[7] P. Bielik, V. Raychev, and M. T. Vechev, "Scalable race detection for Android applications," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, Pittsburgh, PA, USA. New York, NY, USA: ACM, 2015, pp. 332–348.
[8] X. Fu, D. Lee, and C. Jung, "nAdroid: Statically detecting ordering violations in Android applications," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Vösendorf / Vienna, Austria. New York, NY, USA: ACM, Feb. 2018, pp. 62–74.
[9] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for Android applications through refactoring," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Hong Kong, China. New York, NY, USA: ACM, Nov. 2014, pp. 341–352, doi: 10.1145/2635868.2635903.
[10] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, Ambleside, United Kingdom. New York, NY, USA: ACM, 2012, pp. 267–280, doi: 10.1145/2307636.2307661.
[11] Y. Hu and I. Neamtiu, "Static detection of event-based races in Android apps," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, Williamsburg, VA, USA, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 257–270.
[12] N. Salehnamadi, A. Alshayban, I. Ahmed, and S. Malek, "ER Catcher: A static analysis framework for accurate and scalable event-race detection in Android," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Melbourne, Australia. Piscataway, NJ, USA: IEEE Press, 2020, pp. 324–335.
[13] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for Android applications," in *ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, Edinburgh, U.K. New York, NY, USA: ACM, 2014, pp. 316–325, doi: 10.1145/2594291.2594311.
[14] "Github - z3prover/z3: The z3 theorem prover," 2023. [Online]. Available: https://github.com/z3prover/z3
[15] "Roma," 2024. [Online]. Available: https://roma0216.github.io/Roma/
[16] "Processes and threads overview—Android developers," 2021. [Online]. Available: https://developer.android.google.cn/guide/components/processes-and-threads
[17] "Critical capabilities for cloud AI developer services," 2021. [Online]. Available: https://www.gartner.com/en/documents/3999739
[18] "Github - skylot/jadx: Dex to Java decompiler," 2004. [Online]. Available: https://github.com/skylot/jadx
[19] "Download APK fast, free and safe on Android," 2024. [Online]. Available: https://apkpure.com/
[20] S. Arzt et al., "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, Edinburgh, U.K., New York, NY, USA: ACM, 2014, pp. 259–269, doi: 10.1145/2594291.2594299.
[21] J. Wang et al., "AATT+: Effectively manifesting concurrency bugs in Android apps," *Sci. Comput. Program*, vol. 163, pp. 1–18, Mar. 2018, doi: 10.1016/j.scico.2018.03.008.
[22] Q. Li, Y. Jiang, J. Ma, X. Ma, and J. Lu, T. Gu, C. Xu "Effectively manifesting concurrency bugs in Android apps," in *Proc. 23rd Asia-Pacific Softw. Eng. Conf. (APSEC)*, Hamilton, New Zealand, A. Potanin, G. C. Murphy, S. Reeves, and J. Dietrich, Eds. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2016, pp. 209–216, doi: 10.1109/APSEC.2016.038.
[23] T. Su et al., "Guided, stochastic model-based GUI testing of Android apps," in *Proc. 11th Joint Meeting Foundations Softw. Eng. (ESEC/FSE)*, Paderborn, Germany. New York, NY, USA: ACM, 2017, pp. 245–256, doi: 10.1145/3106237.3106298.
[24] T. Gu et al., "Practical GUI testing of Android applications via model abstraction and refinement," in *Proc. 41st Int. Conf. Softw. Eng. (ICSE)*, Montreal, QC, Canada. New York, NY, USA: ACM, 2019, pp. 269–280, doi: 10.1109/ICSE.2019.00042.
[25] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: Generating high-quality test inputs for Android apps via use

case combinations," in *Proc. 42nd Int. Conf. Softw. Eng.*, Seoul, South Korea. New York, NY, USA: ACM, 2020, pp. 469–480, doi: 10.1145/3377811.3380382.

[26] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA), Virtual Event, USA*, New York, NY, USA: ACM, 2020, pp. 153–164, doi: 10.1145/3395363.3397354.

[27] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, "Detecting event anomalies in event-based systems," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, Bergamo, Italy, E. D. Nitto, M. Harman, and P. Heymans, Eds. New York, NY, USA: ACM, 2015, pp. 25–37.

[28] S. Carr, J. Mayo, and C.-K. Shene, "Race conditions: A case study," *J. Comput. Sci. Coll*, vol. 17, no. 1, pp. 90–105, Oct. 2001.

[29] R. H. B. Netzer and B. P. Miller, "What are race conditions? Some issues and formalizations," *LOPLAS*, vol. 1, no. 1, pp. 74–88, 1992, doi: 10.1145/130616.130623.

[30] T. Farah, R. Shelim, M. Zaman, M. M. Hassan, and D. Alam, "Study of race condition: A privilege escalation vulnerability," *Systemics, Cybern. Inform.*, vol. 16, no. 1, pp. 22–26, 2017.

[31] K. Suk Lhee and S. J. Chapin, "Detection of file-based race conditions," *Int. J. Inf. Secur.*, vol. 4, no. 1–2, pp. 105–119, 2004.

[32] M. Bishop and M. Dilger, "Checking for race conditions in file accesses," *Comput. Syst.*, vol. 9, no. 2, pp. 131–152, 1996. [Online]. Available: http://www.usenix.org/publications/compsystems/1996/spr_bishop.pdf

[33] E. Tsyrklevich and B. Yee, "Dynamic detection and prevention of race conditions in file accesses," in *Proc. 12th USENIX Secur. Symp.*, Washington, D.C., USA. USENIX Assoc., 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/dynamic-detection-and-prevention-race-conditions-file

[34] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, Vancouver, British Columbia, Canada. New York, NY, USA: ACM, 2000, pp. 219–232, doi: 10.1145/349299.349328.

[35] C. Flanagan and S. N. Freund, "Detecting race conditions in large programs," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, Snowbird, Utah, USA. New York, NY, USA: ACM, 2001, pp. 90–96, doi: 10.1145/379605.379687.

[36] D. R. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *Proc. 19th ACM Symp. Operating Syst. Princ. (SOSP)*, Bolton Landing, NY, USA. New York, NY, USA: ACM, 2003, pp. 237–252, doi: 10.1145/945445.945468.

[37] N. Licker and A. Rice, "Detecting incorrect build rules," in *Proc. 41st Int. Conf. Softw. Eng., (ICSE)*, Montreal, QC, Canada. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1234–1244, doi: 10.1109/ICSE.2019.00125.

[38] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, Beijing, China, J. Vitek, H. Lin, and F. Tip, Eds., New York, NY, USA: ACM, 2012, pp. 251–262, doi: 10.1145/2254064.2254095.

[39] V. Raychev, M. T. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, Indianapolis, IN, USA, A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds., New York, NY, USA: ACM, 2013, pp. 151–166, doi: 10.1145/2509136.2509538.