Automata-based Trace Analysis for Aiding Diagnosing GUI Testing Tools for Android

Enze Ma[1], Shan Huang[1], Weigang He[1], Ting Su[1]
Jue Wang[2], Huiyu Liu[1], Geguang Pu[1], Zhendong Su[3]
[1]Shanghai Key Lab of Trustworthy Computing, Software Engineering Institute, East China Normal University, China
[2]State Key Lab for Novel Software Tech. and Dept. of Computer Sci. and Tech., Nanjing University, China
[3]Department of Computer Science, ETH Zurich, Switzerland

ABSTRACT

Benchmarking software testing tools against known bugs is a classic approach to evaluating the tools' bug finding abilities. However, this approach is difficult to give some clues on the tool-missed bugs to aid diagnosing the testing tools. As a result, heavy and ad hoc manual analysis is needed. In this work, in the setting of GUI testing for Android apps, we introduce an *automata-based trace analysis* approach to tackling the key challenge of manual analysis, *i.e.*, how to analyze the lengthy event traces generated by a testing tool against a missed bug to find the clues. Our *key* idea is that, we model a bug in the form of a finite automaton which captures its bug-triggering traces; and match the event traces generated by the testing tool (which misses this bug) against this automaton to obtain the clues. Specifically, the clues are presented in the form of three designated automata-based coverage values. We apply our approach to enhance Themis, a representative benchmark suite for Android, to aid diagnosing GUI testing tools. Our extensive evaluation on nine state-of-the-art GUI testing tools and the involvement with several tool developers shows that our approach is *feasible* and *useful*. Our approach enables Themis+ (the enhanced benchmark suite) to provide the clues on the tool-missed bugs, and *all* the Themis+'s clues are identical or useful, compared to the manual analysis results of tool developers. Moreover, the clues have helped find several tool weaknesses, which were unknown or unclear before. Based on the clues, two actively-developing industrial testing tools in our study have quickly made several optimizations and demonstrated their improved bug finding abilities. *All* the tool developers give positive feedback on the usefulness and usability of Themis+'s clues. Themis+ is available at *https://github.com/DDroid-Android/home*.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

*Enze Ma and Shan Huang contributed equally to this work, and they are co-first authors. Ting Su is the corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616361

KEYWORDS

Android GUI Testing, Runtime Verification, Trace Analysis

ACM Reference Format:
Enze Ma, Shan Huang, Weigang He, Ting Su, Jue Wang, Huiyu Liu, Geguang Pu, and Zhendong Su. 2023. Automata-based Trace Analysis for Aiding Diagnosing GUI Testing Tools for Android. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616361

1 INTRODUCTION

In our community, *benchmarking* software testing tools against a set of representative, ground-truth bugs (*e.g.*, Defects4J [33], Lava [16], Magma [29]) is the well-justified and widely-used approach to evaluating and improving the tools' bug finding abilities [34]. Specifically, in the field of GUI testing for Android apps [35, 58], a proliferation of automated testing tools have been developed to help find crash bugs in the apps [17, 27, 36, 37, 39, 45, 53, 60]. However, a recent study [54] benchmarks several such testing tools against a set of real-world bugs. It reveals that these tools miss 53~71% of the bugs — the tool effectiveness gap for finding real-world bugs is large.

In such a situation, the users of a benchmark suite (*e.g.*, the testing tools' developers) likely raise the question "*why does the tool miss these bugs?*" in hope of knowing some clues of potential tool weaknesses for improvement. However, the classic benchmarking approach falls short in such a situation because it can *only* tell the false negatives (*i.e.*, which bugs were missed) *without* any explanation. This shortcomings limits the advantages of benchmarking.

A real example. Figure 1(a) shows a real crash bug (Issue #114 [42]) of `ScarletNotes` [41], an app used to take notes and to-do lists. Figure 1(a) shows this issue's *minimal* bug-triggering trace, which includes five *pivot*[1] input events (steps): (1) c_1: clicking the *notebook creation* button (located at the bottom right on page l_0) to create a notebook (*e.g.*, named as "Notebook1"); (2) c_2: clicking the created notebook "Notebook1" on page l_1 to enter into its directory; (3) c_3: opening the menu by clicking the menu button (located at the bottom left on page l_2); (4) c_4: choosing the "Locked" option on the menu to show the locked notes (page l_3); and (5) c_5: clicking the "×" button on page l_4 to exit from "Notebook1". Note that the bug-triggering condition of this issue is filtering the locked notes under some notebook's directory (*i.e.*, "Notebook1" in this case) *and* then clicking the "×" button to exit from that directory — it does

[1]A pivot event is a necessary event for bug triggering. If a pivot event is removed from the bug-triggering trace, the bug cannot be successfully reproduced.

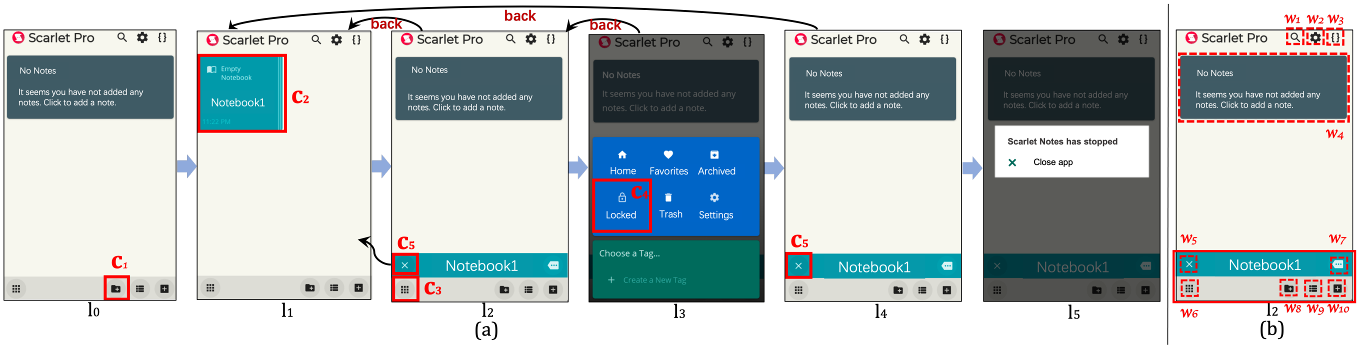

Figure 1: (a) Bug-triggering event trace for `ScarletNotes`'s Issue #114, and (b) list of executable widgets on the GUI page l_2.

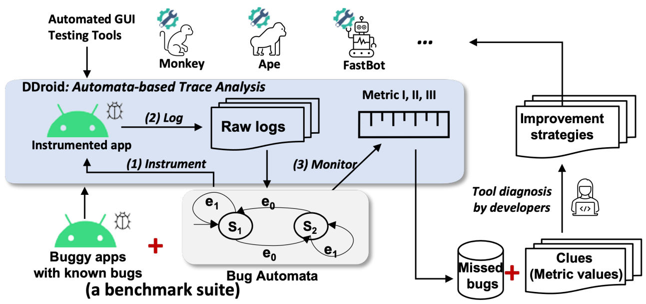

Figure 2: The benchmark suite Themis⁺ enhanced by our automata-based trace analysis approach.

not matter which notebook's directory we are in or whether there exist some locked notes under that directory. By replicating the aforementioned benchmarking study [54] (detailed in Section 4), we find that all of the evaluated testing tools missed this crash bug. In such a situation, the classic benchmarking approach cannot give any clue on this missed bug for aiding diagnosing the testing tools.

Difficulties of current practice. Due to the preceding limitation, tool developers have to *manually* debug their tools against the missed bugs to find some clues. To understand the current practice, we interviewed several tool developers by asking "*what kinds of clues you are looking for during analysis? what difficulties you have in finding these clues?*". *All* the developers responded that they hope to find the clues indicating potential tool weaknesses (*e.g.*, which events cannot be exercised, which UI pages cannot be reached). *However, the main challenge of finding such clues is analyzing the lengthy event traces generated by a tool against its missed bugs.* It makes the manual analysis process time-consuming and difficult.

Specifically, we take Fastbot [21], a popular industrial GUI testing tool in our study, as an example to illustrate the typical manual analysis of tool developers. In face of the missed bug (`ScarletNotes`'s issue #114), the Fastbot's developer *manually* checks whether each pivot UI event of the bug-triggering trace (*i.e.*, c_1, c_2, c_3, c_4, c_5) is *executable*: the developer navigates the app to specific screen pages (*i.e.*, l_0, l_1, l_2, l_3, l_4), dumps the UI layouts, and checks whether the UI widgets (corresponding to c_1, c_2, c_3, c_4 and c_5) are `clickable`. In this case, the developer finds that all the UI widgets are clickable, which means all the pivot UI events could be exercised in theory. However, this clue cannot help explain why the bug was missed. To this end, the developer runs Fastbot on the app (*e.g.*, one hour or more) to *manually* analyze the actual tool behaviors against the missed bug (*e.g.*, analyzing whether Fastbot can indeed reach the pages like l_0, l_1, l_2, l_3 and l_4, *and* exercise c_1, c_2, c_3, c_4 and c_5 in the right order by its testing strategy). Unfortunately, this manual analysis process is time-consuming and difficult because testing tools like Fastbot may generate a large number of random (usually fast-executing) input events (including the pivot events and many irrelevant ones) during testing, although it may sometimes help. For example, Fastbot could generate about 10,000 input events within one hour of testing. It is difficult for human to find the clues by *manually* analyzing such lengthy UI-based event traces. In this case, Fastbot's developer spent more than 3 hours (not including the tool's running time) in finding the clues before giving up. Even worse, this manual analysis becomes more overwhelming when the developer needs to analyze a number of missed bugs or debug different tool versions. When the developer fails to find the clues, they may lose the opportunities for tool improvement.

Our approach and its novelty. The *key* problem in our setting is *how to automatically and effectively analyze the lengthy event traces generated by a tool* (i.e., *the actual tool behaviors) against a missed bug to find the clues.* To this end, our key insight is to cast this challenging problem into *automata-based trace analysis*. Specifically, our idea is to model a bug in the form of an nondeterministic finite automaton (named as the *bug automaton*), which captures its bug-triggering traces. In this way, we can *automatically* match the event trace generated by the testing tool (which misses this bug) against this automaton to monitor tool behaviors. When the event trace cannot be accepted by the automaton (*i.e.*, the bug is missed), we can analyze the matching results of the automaton to obtain the clues. This *automata-based trace analysis* approach tackles the painpoint of manual analysis and is applicable to any off-the-shelf GUI testing tool without tool modifications. At the high-level, our idea can be viewed as the adaption of runtime verification (RV) [7] because the automaton can be interpreted as the *specification* of undesired app behaviors. However, applying existing RV techniques in our setting is difficult, which we will discuss in Section 5.

Specifically, inspired by the clues concerned by tool developers in the interviews and the classic conception of code coverage [68], we introduce three automata-based coverage metrics, *i.e.*, *event coverage*, *event-pair coverage* and *trace-based minimal distance* (detailed in Section 3.3), as the proxies of our clues — *the values of these coverage metrics are the clues provided by our approach.* We also compute some supplementary clues (*e.g.*, the execution times of events and event-pairs). The novelty is that these clues offer the

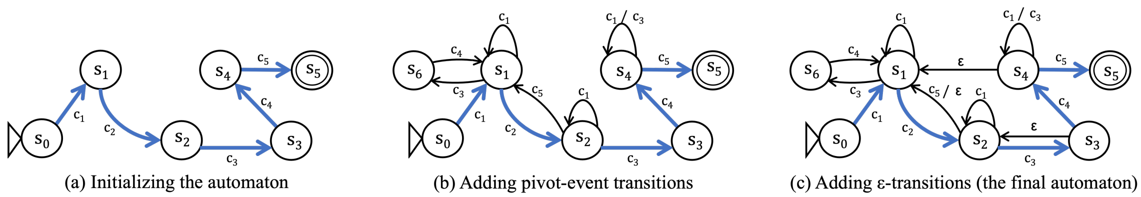

Figure 3: Constructing the bug automaton for `ScarletNote`'s Issue #114 in three steps (a), (b) and (c). (c) is the final bug automaton.

tool developers systematic, fine-grained insights on the potential tool weaknesses, which are difficult to be achieved by the ad-hoc manual analysis (demonstrated by our evaluation in Section 4.4).

Application scenario of our approach. The main application scenario of our approach is to enhance a benchmark suite, thus improving the classic benchmarking. Figure 2 shows the benchmark suite enhanced by our approach (denoted by the blue box, detailed in Section 3). Specifically, we provide each bug with its bug automaton (denoted by the grey box). In this way, given a testing tool under evaluation, the benchmark suite can report the missed bugs *as well as* the clues on these missed bugs. As the users of a benchmark suite, tool developers can inspect the clues (with the bug and the app) to diagnose and improve their tools. Moreover, this benchmark suite can routinely serve as a "regression test suite" for validating the effectiveness of testing tools whenever the tools are modified. It can further reduce the repetitive manual analysis cost of tool developers. Note that the bug automata in our work are *manually* constructed with *a one-time effort*. We will explain how to construct the automata in Section 3.2, and give more discussions in Section 4.6.

Evaluation and Results. We implement a tool named DDroid to support the automata-based trace analysis approach. To evaluate the usefulness, we integrate DDroid into Themis [54], a representative benchmark suite with diverse types of real-world bugs for Android. We named the benchmark suite enhanced by DDroid (and the bug automata) as Themis$^+$. Specifically, we use Themis$^+$ to evaluate nine automated testing tools for Android with *different* testing strategies and implementations, including Ape [27], ComboDroid [60], Stoat [53], DroidBot [36], Humanoid [37], Q-testing [45], Google's Monkey [26], ByteDance's Fastbot [10, 21], and WCTester [66, 67] from Tencent's WeChat team. These tools represent the state-of-the-art and state-of-the-practice.

Our evaluation shows that our automata-based trace analysis approach is *feasible* and *useful*. First, it enables Themis$^+$ to provide the clues on the missed bugs (see Section 4.3), which cannot be achieved by the classic benchmarking (*i.e.*, Themis). Second, *all* the Themis$^+$'s clues are either identical or useful, compared to those manually found by tool developers without any misleading information. The clues have also helped developers successfully pinpoint several tool weaknesses, which were unknown or unclear before. *All* the tool developers explicitly stated that they would enhance their tools based on the clues. Specifically, the two actively-developing industrial testing tools, Fastbot and WCTester, have quickly made several optimizations, and already demonstrated their improved bug finding abilities (Section 4.4). A further interview with the tool developers reveals that *all* the developers are positive on the usefulness and usability of Themis$^+$'s clues.

To sum up, our work has made the following contributions:

- We introduce an automata-based trace analysis approach in the context of GUI testing to enhance the classic benchmarking by providing the clues of tool weaknesses on the missed bugs.
- We introduce three automata-based coverage metrics as the basis of the clues, which can give systematic, fine-grained insights on the potential tool weaknesses.
- Our evaluation shows that the benchmark suite enhanced by our approach is effective and useful. The clues have helped find several tool weaknesses and improved some testing tools.

2 ILLUSTRATIVE EXAMPLE

We use `ScarletNotes`'s Issue #114 (discussed in Section 1) to illustrate our approach.

2.1 Bug Automaton

A bug automaton is represented in the form of a nondeterministic finite automaton with ϵ-transitions (ϵ-NFA [30]). Intuitively, such a bug automaton captures different (non-)minimal bug-triggering traces of the bug. Figure 3(c) gives the automaton of `ScarletNotes`'s Issue #114, in which each node (*e.g.*, $s_0, s_1, s_2, s_3, s_4, s_5, s_6$) denotes an abstract program state, and each transition (*e.g.*, c_1, c_2, c_3, c_4, c_5) denotes an event connecting two states. For example, the trace in blue $[c_1, c_2, c_3, c_4, c_5]$ corresponds to the bug's *minimal* bug-triggering trace. Specifically, s_0 (the initial state) abstracts (and corresponds to) l_0 in Figure 1(a) (denoting the initial state in which no notebook is created), s_1 abstracts l_1 (denoting the state in which some notebook is created) after c_1 is executed, s_2 abstracts l_2 (denoting the state in which the directory of some notebook is opened) after c_2 is executed, s_3 abstracts l_3 (denoting the state in which the menu is shown under the directory of some notebook) after c_3 is executed, s_4 abstracts l_4 (denoting the state in which the locked notes is filtered under the directory of some notebook) after c_4 is executed, and s_5 (the final state) denotes the crashing state after c_5.

For another example, according to the app feature (see Figure 1(a)), one can click the "×" button on l_2 (similar to c_5 on l_4) to return back to l_1, so the automaton includes the transition from s_2 to s_1 (denoted by c_5). This transition helps capture such (non-minimal) traces as $[c_1, c_2, c_5, c_2, c_3, c_4, c_5]$. Additionally, one can press `Back` on page l_2, l_3 or l_4 to jump back to l_1, l_2 or l_1, respectively (denoted by the curved black lines in Figure 1(a)). As a result, one can take some non-minimal traces, *e.g.*, $[c_1, c_2, \texttt{Back}, c_2, c_3, c_4, c_5]$, $[c_1, c_2, c_3, \texttt{Back}, c_3, c_4, c_5]$ or $[c_1, c_2, c_3, c_4, \texttt{Back}, c_2, c_3, c_4, c_5]$ to trigger the bug. To capture such traces, the bug automaton also includes these transitions enabled by `Back`, *i.e.*, the transitions (denoted by ϵ) from s_2 to s_1, s_3 to s_2, and s_4 to s_1. Specifically, ϵ denotes those events like `Back` which are not pivot for bug-triggering but could help capture other non-minimal bug-triggering traces. We will define the bug automaton and explain the construction method in Section 3.2.

Table 1: Clues for WCTester and Fastbot on ScarletNotes's Issue #114 in a simplified textual report. Note that (0) indicates the event or event pair is missed by the tools.

	WCTester	Fastbot
Event Coverage	2/5 (40.0%)	5/5 (100%)
Event-Pair Coverage	4/24 (16.7%)	17/24 (70.8%)
Minimal Distance	3	1
Details of EC Event (Executed_Times)	c_1 (27), c_2 (35), c_3 (0), c_4 (0), c_5 (0)	c_1 (81), c_2 (107), c_3 (17), c_4 (6), c_5 (6)
Details of EPC Event_Pair (Executed_Times)	(c_1,c_2) (14), (c_2,c_3) (0), (c_3,c_4) (0), (c_4,c_5) (0), ...	(c_1,c_2) (50), (c_2,c_3) (11), (c_3,c_4) (4), (c_4,c_5) (0), ...
Details of MD	$[c_1, c_2]$ is covered	$[c_1, c_2, c_3, c_4]$ is covered

2.2 Clues Provided by Our Approach

The clues are presented in the form of the three automata-based coverage values. Table 1 shows the clues for WCTester and Fastbot on the missed ScarletNotes's bug in the form of a simplified textual coverage report, which we explain as follows.

Clue I: Event Coverage (EC). The event coverage tells which pivot events for bug-triggering are covered or missed by a testing tool. Table 1 shows that WCTester misses the three events c_3, c_4 and c_5 (2/5=40% event coverage), while Fastbot covers all the five pivot events (5/5=100% event coverage) but still misses the bug.

Clue II: Event-Pair Coverage (EPC). The event-pair coverage tells which event-pairs are covered or missed by a testing tool. The intuition is that bug finding requires covering the pivot events but also specific event-pairs. For example, (c_1, c_2), (c_2, c_3), (c_3, c_4) and (c_4, c_5) are some typical event-pairs of interest on the minimal bug-triggering trace in Figure 1. Table 1 shows that WCTester and Fastbot achieve 16.7% and 70.8% event-pair coverage, respectively. For example, Fastbot misses the event-pair (c_4, c_5).

Clue III: Trace-based Minimal Distance (MD). The trace-based minimal distance tells how close a testing tool can reach a bug (*e.g.*, which UI pages on the bug-triggering trace can be reached). It uses the number of pivot events to characterize the distance. The smaller the distance is, the closer the tool reaches the bug. When one bug-triggering trace is covered, the distance should be 0. Table 1 shows that WCTester's minimal distance is 3 (*i.e.*, WCTester can only cover $[c_1, c_2]$ in order), while Fastbot's minimal distance is 1 (*i.e.*, Fastbot can cover $[c_1, c_2, c_3, c_4]$ in order but the last event c_5).

Note that in practice Themis+ *visualizes* the clues in the textual report based on the UI transition graph of the missed bug like Figure 1 instead of the bug automaton to ease user understanding and inspection (see an example of the visualized clues at [14]).

2.3 Diagnosing Tools based on the Clues

We present the clues in Table 1 to the developers for tool diagnosis. Based on the clues, WCTester's developer *quickly* locates the suspicious events (*i.e.*, the missed events c_3 and c_5) for diagnosing. First, he inspects the widget properties of c_3 and c_5 (presented by Themis+), and finds that c_3 and c_5 are executable. Next, he inspects the widget types of c_3 and c_5 and finds the root cause, *i.e.*, WCTester fails to support ViewGroup, the widget type of c_3 and c_5. After he fixed this tool weakness, WCTester can find this bug.

Based on the clues, Fastbot's developer *quickly* knows that the tool misses c_5 on page l_4 (as the minimal distance is 1) but executes c_5 on page l_2 (as c_5 is covered). Note that l_2 and l_4 contain c_5 (see Figure 1). Specifically, Fastbot executes c_5 (on l_2) by *only* 6 times (while ComboDroid and Ape executes c_5 on l_2 by 55 and 49 times within the same testing time, respectively). Note that the execution times of events and event-pairs are recorded as the supplementary clues (detailed in Section 3.3). Based on these clues, the developer *quickly* suspects why c_5 is seldom executed and locates the pivot page l_2 for diagnosing. Figure 1(b) shows all the executable widgets on l_2 in the dotted boxes. The developer notes that the six widgets ($w_5 \sim w_{10}$) on l_2, including the widget of c_5 (*i.e.*, the "×" button w_5), have the same widget property values (*i.e.*, the same widget type and resource id). As a result, Fastbot assumes that these six widgets are of the same functional purpose and clusters them into *a widget group* to reduce UI exploration space. In particular, each widget in this group is *purely randomly* selected for execution. But this widget group and the other four widgets on l_2 (*i.e.*, $w_1 \sim w_4$) are selected at the same level. As a result, the probability of executing c_5 on l_2 is $\frac{1}{5} \times \frac{1}{6} \approx 0.03$, which is rather small. The probability of executing c_5 on l_4 is much smaller than 0.03 because c_5 (on l_4) is executed after c_4. This explains why Fastbot misses the event-pair (c_4, c_5). The developer confirmed that this is a design defect in the tool's event selection strategy, and fixed it by prioritizing the widgets (in a clustered group) which have not yet been executed before. The enhanced Fastbot can find this bug. We can see that the clues provide systematic, fine-grained insights to aid diagnosing testing tools, which are hard to be achieved by the manual analysis.

3 APPROACH AND IMPLEMENTATION

3.1 Problem Definition

An Android app is a GUI-based event-driven program P. Each of its screen pages is a GUI layout ℓ (*i.e.*, a GUI tree). Each node of this tree is a GUI view (or widget) w. A GUI event $e = (t, w, o)$ is a triplet, in which $e.t$ denotes its event type (*e.g.*, click, edit), $e.w$ is the widget on which e is executed, and $e.o$ denotes the optional data associated with e (*e.g.*, a string input by edit).

Definition 3.1. ***An event trace***. An event trace T is a sequence of events, which is denoted as $T = [e_1, \ldots, e_i, \ldots, e_n]$, where e_i is an event. When T is executed on an app P, we can obtain a sequence of GUI layouts L, *i.e.*, $L = [\ell_0, \ldots, \ell_{i-1}, \ell_i, \ldots, \ell_n]$, where ℓ_0 is the layout of the app starting page, and ℓ_i is the layout due to the execution of e_i on ℓ_{i-1} ($1 \le i \le n$). Intuitively, the execution of an event trace T can be represented as: $\ell_0 \xrightarrow{e_1} \ell_1 \ldots \ell_{i-1} \xrightarrow{e_i} \ell_i \ldots \ell_{n-1} \xrightarrow{e_n} \ell_n$.

The main goal of an automated GUI testing tool Γ is to find potential crash bugs by generating an event trace T interacting with an app P. Based on Definition 3.1, given a known crash bug, we can define the bug-triggering trace as follows.

Definition 3.2. ***A crash bug triggering trace***. A crash bug r is a crash-inducing fault of P, and usually manifests itself as a runtime exception. The bug-triggering trace T_r of r is an *event trace*, which can deterministically reproduce r. We denote T_r as $T_r = [e'_1, \ldots, e'_i, \ldots, e'_m]$ (e'_i is an event), and the corresponding GUI layouts of T_r as $L_r = [\ell'_0, \ldots, \ell'_{i-1}, \ell'_i \ldots, \ell'_m]$ (ℓ'_i is the layout).

Definition 3.3. ***A 1-minimal bug-triggering trace***. Given a bug-triggering trace T_r of the bug r, if any single event in T_r is removed,

r cannot be reproduced, we say this trace is *1-minimal*. The events in such a 1-minimal trace are named as *pivot events*.

Without ambiguity, all the bug-triggering traces discussed in this paper are 1-minimal unless we explicitly mentioned.

Example. For the bug in Figure 1(a), the bug-triggering trace is $T_r = [c_1, c_2, c_3, c_4, c_5]$. This trace is 1-minimal and executes $L_r = [l_0, l_1, l_2, l_3, l_4, l_5]$ (l_5 denotes the crashing page).

In practice, r may have multiple bug-triggering traces T_rs with different sets of pivot events (these traces lead to the identical exception stack). Without loss of generality, we assume a crash bug r has one bug-triggering trace T_r in the following definitions, and we discuss the case of multiple bug-triggering traces T_rs later.

Problem Definition. Our problem is, given a crash bug r of P and a testing tool Γ, how to automatically and effectively match the event trace $T = [e_1, \ldots, e_i, \ldots, e_n]$ generated by Γ against r's bug-triggering trace $T_r = [e'_1, \ldots, e'_i, \ldots, e'_m]$ to find the clues.

3.2 Bug Automaton and Its Construction

To tackle the preceding problem, our key idea is that, given a bug r, we construct a *bug automaton* M to represent r based on T_r; and match T against the automaton M to find the clues. Specifically, we formulate M in the form of a nondeterministic finite automaton with ϵ transitions (ϵ-NFA for short) [30].

Definition 3.4. ***Bug Automaton.*** A bug r's automaton M is formulated as a ϵ-NFA. Given r's the minimal bug-triggering trace $T_r = [e'_1, \ldots, e'_i, \ldots, e'_m]$ and its corresponding GUI layouts $L_r = [\ell'_0, \ldots, \ell'_{i-1}, \ell'_i \ldots, \ell'_m]$, we define M as $M = (S, \Sigma, \delta, s_0, F)$, where

- Σ is a finite set of input symbols, *i.e.*, $\Sigma = \{e'_1, \ldots, e'_i, \ldots, e'_m\} \cup \{\epsilon\}$. Here, $e'_1, e'_2, \ldots, e'_m$ are the pivot events on T_r, and ϵ denotes any event (like back in the automaton in Figure 3(c)) which are not pivot for bug-triggering but could lead to other possible non-minimal traces reaching r.
- S is a finite set of abstract program states which can be reached by executing the input symbols in Σ on app P.
- δ is a transition function, *i.e.*, $\delta : S \times \Sigma \longrightarrow \mathcal{P}(S)$, where $\mathcal{P}(S)$ is the power set of S.
- $s_0 \in S$ is the initial state of M.
- F is the set of final states. Specifically, in our setting, F only contains one state which denotes the crashing state.

Bug Automaton Construction Given a bug-triggering trace T_r, we follow three steps to *manually* construct r's bug automaton M. In the following, we use `ScarletNotes`'s Issue #114 (see Figure 1) to illustrate the automaton construction method (shown in Figure 3).

Step 1: Initializing the automaton by the minimal bug-triggering trace. Based on the minimal bug-triggering trace T_r and its corresponding GUI layouts L_r, we can initialize the set of input symbols Σ, the set of abstract program states S, and the transition function δ of the automaton M. Specifically, the GUI layouts L_r are abstracted to the set of states S, *i.e.*, ℓ'_i is abstracted to s_i. Here, ℓ'_0 (the app's starting page) is abstracted to s_0 (M's initial state), and ℓ'_n (the app's crashing page) is abstracted to s_n (M's final state). According to the execution of T_r, if there exists a page transition $\ell'_i \xrightarrow{e_{i+1}} \ell'_{i+1}$, the corresponding state transition $s_i \xrightarrow{e_{i+1}} s_{i+1}$ will be added into the transition function δ. In this way, the initial bug automaton is constructed, *i.e.*, $s_0 \xrightarrow{e_1} s_1 \ldots s_{i-1} \xrightarrow{e_i} s_i \ldots s_{n-1} \xrightarrow{e_n} s_n$. Take `ScarletNotes`'s bug in Figure 1 as example, based on its T_r, we can decide that $\Sigma = \{c_1, c_2, c_3, c_4, c_5\}$, $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$, s_0 and s_5 are the initial and final state, respectively, and the initial transition function δ corresponding to the transitions in Figure 3(a).

Step 2: Adding other transitions enabled by the pivot events. After **Step 1**, the automaton only captures the minimal bug-triggering trace. To capture those non-minimal bug-triggering traces, we need to include other transitions enabled by the pivot events into the automaton. To this end, we check whether any pivot event in Σ can be executed on each state in S (except s_0 and s_n) and lead to new transitions and/or new states. We will add any new transition and/or state into the automaton, and apply the same checking process on the new states until no new transitions or states can be found. Let us take s_1 in the automaton in Figure 3(a) as an example to enumerate the input symbols in Σ against s_1. According to the app feature, (1) s_1 can take c_1 to reach s_1 itself because we can execute c_1 on s_1 (corresponding to l_1) to create some notebook (recall that s_1 denotes the abstract state in which some notebook is created); (2) s_1 can take c_2 to reach s_2 according to T_r (already included in the automaton); (3) s_1 can take c_3 to reach s_6, a new abstract state denoting the menu is shown on top of the app's main page, which is different from s_3 (because s_3 denoting the menu is shown under the directory of some notebook); (4) s_1 cannot take c_4 and c_5 because c_4 and c_5 do not exist on s_1 (corresponding to l_1). As a result, we added all the transitions enabled by the pivot events for s_1. Similarly, we can enumerate the input symbols in Σ against the remaining states in S. After this step, we obtain the automaton shown in Figure 3(b). The automaton captures those non-minimal bug-triggering traces like $[c_1, c_2, c_5, c_2, c_3, c_4, c_5]$.

Step 3: Adding the ϵ-transitions. In addition to the pivot events, one may take some non-pivot events (like Back) to reach r. Thus, we annotate such events as ϵ (at this time Σ is updated to $\{c_1, c_2, c_3, c_4, c_5, \epsilon\}$) and include the transitions enabled by ϵ. For example, according to the app feature (see Figure 1(a)), one can press Back on page l_2, l_3 or l_4 to jump back to l_1, l_2 or l_1, respectively (denoted by the curved black lines in Figure 1(a)). Thus, we add the ϵ-transitions from s_2, s_3 and s_4 to s_1, s_2 and s_1, respectively. In this way, the automaton can capture such new non-minimal bug-triggering traces $[c_1, c_2, \text{Back}, c_2, c_3, c_4, c_5]$, $[c_1, c_2, c_3, \text{Back}, c_3, c_4, c_5]$ or $[c_1, c_2, c_3, c_4, \text{Back}, c_2, c_3, c_4, c_5]$. After this step, we obtain the final automaton shown in Figure 3(c).

Discussion. (1) *Handling multiple bug-triggering traces.* A crash bug r may have multiple bug-triggering traces T_rs with different sets of pivot events. In such cases, by Definition 3.4, we construct an ϵ-NFA based on each bug-triggering trace T_r, and merge these ϵ-NFAs together into a new ϵ-NFA by connecting their initial and final states with ϵ. (2) *The bug-triggering trace T_r should be 1-minimal.* Because such a bug-triggering trace makes M expressive, succinct and precise. If T_r includes non-pivot events, M could become unnecessarily complicated and may lead to misleading clues. For example, assuming event e_i is an non-pivot event but included in T_r, if a testing tool triggers r but does not cover e_i, the clue that e_i is not covered does not make sense. In practice, given a bug-triggering trace, we manually reduce it to a 1-minimal one (removing one event at one time and then checking whether the bug can be reproduced). It takes little effort because the bug-triggering traces obtained from bug reports are already or close to 1-minimal. (3) Given all the bug-triggering traces T_rs, *the bug automaton is precise and complete by construction*. Section 4.6 empirically validates the precision and completeness of the manually constructed automata.

After the construction, we automatically convert M in the form of ϵ-NFA to its equivalent deterministic finite automaton (DFA) by eliminating the ϵ transitions. Formally, the DFA M_d is $M_d = (S_d, \Sigma_d, \delta_d, q_0, F)$, where all the components have their similar interpretations as for the ϵ-NFA and $\Sigma_d = \Sigma \setminus \{\epsilon\}$. We conduct this conversion because the DFA (without ϵ-transitions) is algorithmically more convenient for defining and computing the coverage metrics (detailed in Section 3.3). Note that (1) M and M_d are equivalent and accept the same language [30], so it is *safe* to match T (which only contains the symbols in Σ_d) against M_d. (2) The conversion is not expensive as the sizes of ϵ-NFAs are relatively small. Moreover, ϵ-NFA is more intuitive for human understanding (like the UI transition graph in Figure 1) and easier for manual construction than its equivalent DFA. In Table 3, "ϵ-NFA Sizes" and "DFA Sizes" show the sizes of ϵ-NFA and its DFA, respectively.

3.3 Coverage Metrics based Clues

We introduce three coverage metrics at the automaton level of $M_d = (S_d, \Sigma_d, \delta_d, q_0, F_d)$, the equivalent DFA of the bug automaton M (ϵ-NFA), as the basis of the provided clues.

Clue I: Event Coverage. Let E_a be the set of all the events in Σ_d, and let E_c be the set of events executed by a testing tool Γ. Formula (1) defines *event coverage* (EC) to characterize how many events could be covered by Γ.

$$\mathsf{EC} = |E_c|/|E_a| \times 100\% \tag{1}$$

Conceptually, EC is similar to the *statement coverage* in classic software testing. Since the events in Σ_d are necessary to trigger r, EC can assess the tool effectiveness when Γ cannot execute all these events. The higher EC is, the more likely Γ can find the bug r. If Γ cannot execute some events, it likely indicates some tool weaknesses. For example, as we illustrated in Section 2.2, WCTester misses some events as it fails to support these events' widget type.

Clue II: Event-Pair Coverage. Let I_a be the set of all event pairs (e_x, e_y) in M_d, where e_x and e_y are the events in Σ_d and denote the events of two adjacent transitions in δ_d. For example, in Figure 3(c), (c_4, c_5) is an event-pair as c_4 and c_5 are the events of two adjacent transitions. Formally, $I_a = \{(e_x, e_y) | \forall s_i, s_j, s_k \in S_d . \exists e_x, e_y \in \Sigma_d . \delta_d(s_i, e_x) = q_j \wedge \delta_d(s_j, e_y) = s_k\}$. Let I_c be the set of the covered event pairs. Specifically, we say the event pair (e_x, e_y) is covered if both e_x and e_y are executed in the order of e_x immediately followed by e_y. Formula (2) defines *event-pair coverage* (EPC) to characterize how many event pairs could be covered by Γ.

$$\mathsf{EPC} = |I_c|/|I_a| \times 100\% \tag{2}$$

Conceptually, EPC is similar to the *branch coverage* in classic software testing. EPC is a stronger metric than EC. EPC can (1) assess the ability of a testing tool Γ to execute two adjacent transitions, and (2) reflect the diversity of event traces generated by Γ. The higher EPC is, the more likely Γ can stress test the interactions between pivot events. If some event pairs are not covered, it may indicates some tool weaknesses. For example, as we illustrated in Section 2.2, Fastbot missed the event-pair (c_4, c_5) which indicates some tool weaknesses. Note that this metric is identical to the *event-interaction coverage* in traditional GUI software testing [43] and can be extended to length-n event sequence coverage ($n \geq 2$).

Clue III: Trace-based Minimal Distance. Let $T_{\Sigma_d} = [e_1, \ldots, e_i, \ldots, e_n]$ ($e_i \in \Sigma_d$) be the event trace generated by a testing tool Γ. Let $S_c = \{s_0, \ldots, s_j, \ldots, s_m\}, s_j \in S_d, 1 \leq j \leq m$ be the set of states that T_{Σ_d} can reach when matching T_{Σ_d} against the automaton M_d. Let $distance(s_j, F_d)$ be the minimal number of events (or transitions) required to take from s_j to reach F_d on M_d. Formula (3) defines the trace-based minimal distance (MD) to characterize how close a testing tool Γ can reach a crash bug r.

$$\mathsf{MD} = min(\{distance(s_j, F_d) | s_j \in S_c\}) \tag{3}$$

where $min()$ returns the minimal element of a set.

For example, if $S_c = \{s_0, s_1, s_2, s_6\}$ is the set of states reached by a testing tool on the automaton in Figure 1(c), the value of MD is 3. Because the minimal distance is from s_2 to the final state s_5 by following the three events c_3, c_4, and c_5.

MD assesses the tool effectiveness from the perspective of *path-based testing* in classic software testing. This metric can (1) assess whether a testing Γ can exercise the events of the bug-triggering trace in some specific orders, and (2) quantify how far Γ is to reach r in terms of number of events to be executed. It indicates the ability boundary of a testing tool. If Γ can find the crash bug r, the MD should be 0. MD is a stronger metric than EC and EPC because a tool may achieve 100% EC or EPC but may not achieve MD as 0.

Other clues: Execution times of events and event-pairs. We compute the execution times (ET) of covered events (*i.e.*, the events in E_c of EC) and event-pairs (*i.e.*, the event-pairs in I_c of EPC), respectively, as the supplementary metrics. ET is similar to the *execution count* metric in classic code coverage tools like gcov [22] for performance profiling in terms of statements and branches.

3.4 Implementation

Figure 2 illustrates the workflow of our automata-based trace analysis approach (denoted by the blue box). Specifically, given a bug r of an buggy app P and its automaton M (manually constructed according to the method described in Section 3.2), our approach conducts the following three automated steps to obtain the clues.

(1) Instrumentation. The buggy app P is automatically instrumented at the pivot events in T_r. Let T_r be $[e'_1, \ldots, e'_i, \ldots, e'_m]$, we instrument P at the event listener of each event e'_i. In this way, e'_i will be logged when it is executed by the tool Γ.

(2) Logging. The testing tool Γ is run against the instrumented app P to log the executed pivot events. All the logged pivot events forms an event trace L. Γ is allocated with enough testing time for running to reach the saturation point.

(3) Monitoring. To ease the computation of coverage metrics, we automatically convert the bug automaton M from an ϵ-NFA to an equivalent DFA M_d. Next, we match the logged event trace L against M_d, and compute the coverage metrics (*i.e.*, EC, EPC, MD and ET). During the matching, one event is taken from L at one time and matched against the transitions of M_d, and all the covered events,

Table 2: Selected automated GUI testing tools in our study.

Tool	Venue/Source	Main Testing Strategies
Stoat	ESEC/FSE'17	Model-based
DroidBot	ICSE'17	Model-based
Ape	ICSE'19	Model-based
Humanoid	ASE'19	Deep learning-based
ComboDroid	ICSE'20	Model-based
Q-testing	ISSTA'20	Reinforcement learning-based
Monkey	Google	Random testing
Fastbot	ByteDance	Model- & Reinforcement learning-based
WCTester	WeChat	Random & Reinforcement learning-based

event pairs, the reached states and the execution times are recorded to compute the coverage metrics.

We developed a tool DDroid (written in `Python`, `shell` and `HTML`) to support the application of our approach. We use JFlap [46] and its extension PUTFlap [49] to specify the bug automaton. We use Gradle Transformer [25] and ASM [4, 9] to automatically instrument apps at the event handlers to uniquely log executed events [38, 51]. We use automata-lib [19] to convert an ϵ-NFA to an equivalent DFA, and the Floyd's algorithm [64] to compute the `MD` metric. We visualize the clues via interactive `HTML` pages to ease user inspection.

4 EMPIRICAL EXPERIMENT

4.1 Research Questions

- **RQ1**: Enhanced by the automata-based trace analysis approach, can Themis^{+} provide the clues on the bugs missed by automated GUI testing tools, compared to Themis?
- **RQ2**: How useful are the clues provided by Themis^{+} for aiding diagnosing GUI testing tools, compared to the clues manually found by tool developers based on *only* the missed bugs?
- **RQ3** : How well other alternative trace analysis approaches perform in finding the clues? Can they outperform the automata-based trace analysis approach in finding useful clues?

RQ1 investigates the *feasibility* of the automata-based trace analysis approach to provide some clues on the tool-missed bugs, thus improving the classic benchmarking. **RQ2** investigates the *usefulness* of the automata-based trace analysis approach, *e.g.*, better understanding testing tools' behaviors, diagnosing potential tool weaknesses, and improving the tools' bug finding abilities. **RQ3** investigates the *effectiveness* of the automata-based trace analysis approach compared to other alternative trace analysis approaches, *i.e.*, to what extent our approach is really needed.

4.2 Experimental Setup

Experimental Environment. We deployed our experiment on a 64-bit Ubuntu 20.04 machine (64 cores, AMD 3995WX CPU, and 128GB RAM) and Google Android 7.1 emulators.

GUI testing tools. We selected nine GUI testing tools including six academic ones (Ape [27], ComboDroid [60], Humanoid [37], DroidBot [36], Q-testing [45] and Stoat [53]) and three industrial ones (Monkey [26], Fastbot [10, 21], and WCTester [66, 67]) for our experiment. These tools represent the state-of-the-arts. Note that we used the latest versions of these tools at the time of our study. The academic tools and Fastbot are publicly available on GitHub. Monkey is released with Android SDK. WCTester is obtained on request from WeChat's testing team. Table 2 summarizes these selected tools and their main testing strategies. Readers can refer to these tools' papers for more information. We did not include old tools like Sapienz [39] as it only works old Android versions.

The Benchmark Suite. We applied our approach to Themis [54], a benchmark suite with real-world bugs for Android among others [52, 63]. Themis is *representative* as it contains 52 crash bugs with different complexities from 20 different categories of apps. Each of these bugs is provided with its minimal bug-reproducing traces and the corresponding buggy app version. Interested readers can refer to Table 3 in Themis's paper [54] or Themis's bug repository [55] for bug details. To build Themis^{+} based on Themis, one graduate and one undergraduate students who participated in this research work manually built the bug automaton for each bug. Before constructing the automata, the students spent some time in getting familiar with the apps and the bugs. In our experience, it roughly took 2~20 minutes to build one bug automata depending on the complexity of the bug. It took us about 8 hours in total to build and validate all the bug automata. In this process, we excluded 2 bugs of `WordPress` (because the buggy app versions cannot be compiled anymore due to an obsoleted third-party library), 1 bug of `AmazeFileManager` (which cannot be deterministically reproduced), 1 bug of `Phonograph` (because the bug requires adding 2000 music files, which is unrealistic for automated testing tools), and 1 bug of `Frost-for-Facebook` (avoiding violating Facebook's user policy due to random fuzzing). Thus, we finally got 47 instrumented APKs which can deterministically reproduce the corresponding bugs. Table 3 (column "Bugs") lists these bugs.

Evaluation setup for RQ1. We benchmarked the nine selected testing tools on the 47 bugs to identify missed ones, and computed the automata-based coverage metrics. We followed the instructions of Themis [55] (see Section 3.3 in [54]) to run these tools: each tool is run against each bug on one emulator in one run; each run was allocated with 6 hours for thorough testing, and repeated 5 times to mitigate the randomness. For the nine selected tools, the whole evaluation took about 47×6×5×9 = 12,690 machine hours.

Evaluation setup for RQ2. We invited the developers of seven tools (listed in Table 4's column "Tool") to investigate the usefulness of the Themis^{+}'s clues. Monkey and Q-testing were excluded because Monkey's and Q-testing's developers did not reply to our invitation. We find that six of these seven tools (except Fastbot) are developed and maintained by *only* one person, respectively. In this case, it is difficult to involve different developers per tool to conduct the study with statistical tests. Therefore, we involved 7 developers (one developer per tool) in the study and designed a rigorous two-step study which we believe is already enough and valid to answer RQ2. In the first step, we gave the tool developers the missed bugs (*i.e.*, the output of Themis) and let them try their best to manually find the clues based on the buggy app and the bug-triggering traces. The developers followed the similar manual analysis process described in Section 1 (*e.g.*, running their tools against the missed bugs) to find the clues without time limits. This step aims to obtain the "ground-truth" clues of developers with their best effort. In the second step, we gave the same developers the Themis^{+}'s clues (*i.e.*, the output of Themis^{+}). The clues are visualized based on the textual coverage report in Table 1 to ease inspection. We let them validate whether the clues are *useful*, *identical*, or *misleading*, compared to their prior own clues. Specifically,

Table 3: Evaluation results of the nine GUI testing tools based on THEMIS$^+$ against the 47 real-world crash bugs.

Bugs	ϵ-NFA	DFA	WCTester			Fastbot			Ape			ComboDroid			Monkey			Stoat			DroidBot			Humanoid			Q-testing		
	Sizes	Sizes	EC	EPC	MD	EC	EPC	MD	EC	EPC	MD	EC	EPC	MD	EC	EPC	MD	EC	EPC	MD	EC	EPC	MD	EC	EPC	MD	EC	EPC	MD
AD-118	13/26	13/117	50	43	7	50	43	7	50	47	7	50	47	7	50	47	7	100	67	0*	88	70	3	100	64	0*	50	36	7
AD-285	9/21	9/63	100	58	0*	50	26	2	67	42	2	83	63	1	67	37	2	67	35	2	67	53	2	50	26	2	83	63	0*
AFM-1232	5/7	5/25	75	83	1	75	83	1	75	83	1	75	83	1	75	83	1	100	83	0*	75	83	1	75	83	1	75	83	1
AFM-1796	4/5	4/16	100	100	0*	100	100	0*	100	100	0*	100	100	0*	67	75	1	100	100	0*	100	100	0*	67	75	1	100	100	0*
AFM-1837	4/5	4/16	33	25	2	67	75	1	67	75	1	67	75	1	100	100	0*	33	25	2	33	25	2	33	25	2	33	25	2
AB-261	14/44	14/140	100	60	6	100	37	0*	78	36	4	78	37	6	78	27	5	67	21	6	67	31	6	89	31	2	66	16	6
AB-375	11/36	13/117	75	41	1	88	62	0*	75	58	3	50	13	4	88	48	1*	75	55	3	62	31	2	75	45	3	50	15	4
AB-480	12/38	13/156	92	46	4*	92	44	4*	100	38	3*	92	37	4*	25	8	4	92	47	1	50	22	4	58	17	4	33	9	4
AB-697	10/17	10/90	63	24	6	63	28	5	88	48	3	75	45	5	88	59	6	38	14	6	63	21	5	75	45	5	37	13	6
AB-703	7/13	7/56	86	50	3	86	58	3	86	63	3	86	63	3	86	58	3	57	21	3	86	50	3	86	54	3	71	29	3
Anki-4200	5/9	5/25	100	100	0*	100	100	0*	100	100	0*	100	100	0*	75	33	3	100	50	0*	100	58	2	100	75	0*	100	75	3
Anki-4451	16/32	22/176	100	68	0*	71	39	2	86	39	1	86	48	1	71	52	2	86	48	1	100	39	0*	100	39	0*	/	/	/
Anki-4707	4/5	4/16	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	80	0*	100	80	0*	100	100	0*	/	/	/
Anki-4977	6/9	6/24	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	86	0*	100	71	0*	100	71	0*	100	86	0*	100	85	0*
Anki-5638	3/3	3/9	50	50	1	50	50	1	50	50	1	50	50	1	50	50	1	50	50	1	50	50	1	50	50	1	50	50	1
Anki-5756	6/13	6/36	100	92	0*	100	77	0*	100	62	0*	100	85	0*	100	46	0*	100	31	0*	100	62	0*	100	69	0*	100	69	0*
Anki-6145	17/39	17/153	75	40	2	63	24	3	63	27	3	75	36	2	75	40	2	63	29	3	63	31	3	63	29	3	63	26	3
APM-116	2/1	2/2	100	-	0*	100	-	0*	100	-	0*	100	-	0*	100	-	0*	100	-	0*	100	-	0*	100	-	0*	100	-	0*
collect-3222	7/15	7/42	100	100	0*	100	74	0*	100	79	0*	100	94	0*	100	84	0*	100	79	0*	100	84	0*	80	26	1	0	0	3
comm-1385	6/11	6/30	100	88	0*	100	63	2	100	69	0*	100	75	1	50	25	2	75	63	1	100	56	2	50	25	2	/	/	/
comm-1391	6/15	6/36	80	65	4	80	61	4	100	73	3	80	60	4	0	0	5	80	60	4	100	61	3	100	91	1	/	/	/
comm-1581	8/15	7/35	20	10	4	20	10	4	20	10	4	20	10	4	80	50	1	20	10	4	20	10	4	20	10	4	/	/	/
comm-2123	5/7	5/25	100	100	0*	100	78	0*	100	100	0*	100	100	0*	100	100	0*	100	78	0*	75	67	1	100	100	0*	/	/	/
comm-3244	7/15	7/42	80	69	1	80	75	1	80	75	1	0	0	5	60	50	2	40	19	3	100	69	0*	/	/	/	/	/	/
FL-4881	4/5	4/16	33	20	2	67	80	1	33	20	2	33	20	2	67	80	1	33	20	2	33	20	2	33	20	2	0	0	3
FL-4942	5/11	5/25	100	100	0*	100	100	0*	100	93	0*	100	60	1	100	80	0*	100	73	0*	100	73	0*	100	87	0*	0	0	4
FL-5085	4/6	4/16	100	100	0*	100	100	0*	100	100	0*	100	100	0*	67	75	1	67	50	1	100	100	0*	67	75	1	0	0	3
GHD-73	3/3	3/9	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*
MFB-224	2/1	2/1	0	-	1	100	-	0*	0	-	1	100	-	0*	0	-	1	0	-	1	0	-	1	0	-	1	0	-	1
NC-1918	3/3	3/9	100	100	0*	0	0	2	100	100	0*	0	0	2	100	100	0*	0	0	2	100	100	0*	100	100	0*	/	/	/
NC-4026	3/3	3/9	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	50	50	1	100	100	0*	/	/	/
NC-4792	6/9	6/36	100	73	2	100	82	2	100	100	0*	80	55	2	60	46	2	80	46	3	40	9	4	100	64	2	/	/	/
NC-5173	5/9	5/35	100	100	0*	67	55	0*	100	100	0*	100	91	0*	67	64	0*	83	82	0*	100	73	0*	100	100	0*	/	/	/
ON-745	8/16	8/88	50	17	2	60	45	0*	60	34	1	60	32	1	60	21	0*	60	19	0*	60	26	0*	60	21	0*	20	5	2
OEAA-2198	5/8	4/20	100	80	0*	100	80	0*	100	80	0*	100	90	0*	100	60	0*	100	50	0*	100	70	0*	100	70	0*	0	0	3
OL-67	3/2	2/2	0	-	2	100	-	0*	100	-	0*	100	-	0*	50	-	1	0	-	2	0	-	2	0	-	2	0	-	2
OE4A-637	13/25	13/91	86	63	3	71	53	3	86	63	3	86	68	1	71	42	4	71	37	3	86	53	2	71	42	3	14	0	5
OE4A-729	6/9	6/36	80	71	1	100	93	0*	100	93	0*	100	100	0*	40	21	3	40	7	4	60	36	3	80	57	1	20	0	4
<u>SN-114</u>	6/19	6/36	40	17	3	100	71	1	100	95	1	100	95	1	20	5	5	60	14	3	100	38	2	60	29	3	0	0	5
SF-239	4/5	4/16	67	75	1	67	75	1	67	75	1	67	75	1	100	100	0*	67	75	1	67	75	1	67	75	1	67	75	1
WP-6530	8/16	8/64	71	55	7	71	55	7	71	50	7	71	50	7	57	15	7	43	5	6	57	25	7	71	35	6	/	/	/
WP-7182	7/11	7/42	60	53	2	60	53	2	60	47	2	60	53	2	60	47	2	60	27	2	60	47	2	60	40	2	/	/	/
WP-10302	3/3	3/9	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	100	0*	0	0	2
WP-10363	4/7	4/20	100	86	0*	100	71	0*	100	43	0*	/	/	/	25	14	1	50	14	0*	75	57	0*	75	71	0*	/	/	/
WP-10547	5/6	5/20	100	100	0*	100	100	0*	67	67	2	/	/	/	67	50	2	100	83	0*	100	83	0*	100	83	0*	/	/	/
WP-11135	4/5	4/16	100	100	0*	100	100	0*	100	80	0*	/	/	/	100	60	0*	100	20	0*	100	80	0*	100	100	0*	/	/	/
WP-11992	5/10	5/25	100	100	0*	100	100	0*	100	100	0*	100	100	0*	100	50	0*	100	29	2	75	14	3	100	79	0*	/	/	/
#Best Values			31	26	27	32	20	30	33	18	29	28	19	25	25	14	25	21	7	25	26	8	22	25	11	25	7	5	8
#Found/Missed			**24 / 23**			**26 / 21**			**24 / 23**			**19 / 25**			**19 / 28**			**20 / 27**			**19 / 28**			**20 / 26**			**6 / 24**		

we say *identical* if developers decided THEMIS$^+$'s clues are identical to their found clues; *useful* if developers decided THEMIS$^+$'s clues provide more useful information for tool diagnosis than their found clues (*e.g.*, the THEMIS$^+$'s clues cannot be found by manual analysis *or* are more precise than the clues found by manual analysis); *misleading* if developers decided THEMIS$^+$'s clues are contradictory *w.r.t.* their found clues. Note that all the involved developers are experts and have been actively maintaining the tools for 3~4 years. Thus, they have enough expertise to evaluate the usefulness of the THEMIS$^+$'s clues. This study was conducted with developers online. After the study, we conducted an interview with each developer to solicit their feedback on THEMIS$^+$'s clues.

Evaluation setup for RQ3. We compared the automata-based trace analysis approach with two simple alternative trace analysis approaches, *i.e.*, *simple trace comparison* (***simple TC*** for short) and *simple runtime verification* (***simple RV*** for short). Specifically, *simple TC* represents a naive trace analysis method. It directly compares the event trace T generated by a testing tool and the bug-triggering trace T_r of a known bug r. It reports *the first differing event between these two traces T and T_r*. *Simple RV* matches the event trace T generated by a testing tool against the constructed bug automaton M. It reports *the first event which cannot be accepted by the automaton M*. Because the clues reported by these two approaches and ours cannot be directly compared. To fairly compare these approaches, we use the *first missed event* in the bug-triggering trace of a missed bug as the comparison metric. Formally, given a bug-triggering trace $T_r = [e'_1, \ldots, e'_i, \ldots, e'_m]$, e'_i is the *first missed event* in T_r if e'_i is missed but all the events $e'_1, \ldots, e'_{i-1}$ are covered in the order by T. Note that *simple RV* used the bug automata constructed by us. Our approach computes the first missed event based on the trace-based minimal distance MD. We used the event traces generated by the testing tools in RQ1 for evaluation.

4.3 Results of RQ1

THEMIS *v.s.* THEMIS$^+$ Table 3 gives the results of RQ1. Column "Bugs" lists the 47 bugs. For example, "SN-114" denotes `ScarletNote`'s Issue #114. In Table 3, the last row "#Found/#Missed" gives the output of THEMIS on these tools in the form of X/Y, where X and Y are the numbers of found and missed bugs, respectively. We can see that THEMIS can only identify the missed (and found) bugs.

With the help of our approach, THEMIS$^+$ can provide the clues on the missed bugs, which cannot be obtained by THEMIS. The columns with tool names (*e.g.*, WCTESTER, FASTBOT) give the achieved *best* coverage values of the three main metrics, *i.e.*, event coverage (EC), event-pair coverage (EPC) and trace-based minimal distance (MD), for each bug/tool among the five independent testing runs. We

Table 4: Validation results of Themis+'s clues *w.r.t.* the clues manually found by tool developers on the missed bugs.

Tool	#Identical	#Useful	#Misleading
WCTester	10	13	0
Fastbot	8	13	0
Ape	8	15	0
ComboDroid	11	14	0
Stoat	10	17	0
DroidBot	13	15	0
Humanoid	11	15	0
Total	71 (41%)	102 (59%)	0

focus on these achieved best values as they indicate the best tool performance. Take the results of WCTester on bug "SN-114" as an example (see row "SN-114" under column "WCTester"), the best achieved EC, EPC, MD among the five testing runs are 40%, 17% and 3, respectively. From such metric values, we can obtain the clues, *e.g.*, *which events and event-pairs are missed* and *how close a tool can reach the bug*. For example, Section 2.3 illustrates the clues on the missed bug "SN-114" for WCTester.

Miscellaneous. In Table 3, symbol "/" denotes the coverage value is unavailable due to tool issues. For example, Q-testing only successfully ran on 29 bugs (we reported the tool issues to Q-testing's developer but did not get reply). Symbol "-" denotes the coverage metric is not applicable. For example, "APM-116" does not have EPC because its bug automaton only has one transition.

4.4 Results of RQ2

How useful are the Themis+'s clues? Table 4 gives the validation results on Themis+'s clues. Column "#Identical", "#Useful" and "#Misleading" denote the numbers of missed bugs for which Themis+ finds the *identical*, *useful*, or *misleading* clues respectively, compared to the clues manually found by tool developers. From Table 4, we find that *all* the Themis+'s clues are identical or useful compared to the clues manually found by developers, without any misleading ones. Specifically, Themis+ provided the *identical* and *useful* clues, respectively, for 71 (41%) and 102 (59%) of the missed bugs for all tools. We provided the detailed validation results on each missed bug per tool in the supplementary material [15].

How can Themis+ find identical or useful clues? In 71 cases, Themis+ can find the identical clues *w.r.t.* the manual analysis of tool developers. For example, "SF-239" requires a multi-touch event on an item list. For the tool missing this bug, the developers can find the clue that the tool cannot emit multi-touch by manual analysis. Themis+ can find the identical clue as EC can tell the multi-touch event is not covered. In 102 cases, Themis+ can find useful clues. Take "NC-4792" as an example, the bug-triggering trace has five events: e_1 (*opening the sidebar navigation drawer*), e_2 (*selecting "Auto upload" in the drawer*), e_3 (*selecting "Remote folder" on the "Auto upload" page*), e_4 (*selecting "New folder" on the main page*), and e_5 (*pressing the "Create" button to create a new folder*). For this bug, WCTester's developer cannot find any clue although he observes that the tool could click all the widgets of $e_1 \sim e_5$. Themis+ finds the clue that WCTester can indeed generate these events (because the EC is 100%) but these events are not executed in the right order (because its MD is 2). Themis+ reveals that WCTester never creates the folder (by e_4 and e_5) after the "Remote folder" option is selected (by e_1, e_2 and e_3). This clue is hard to obtain by manual analysis.

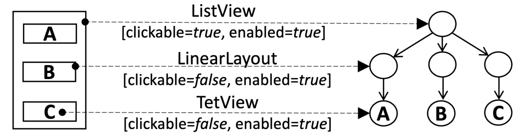

Figure 4: An example of event generation strategy.

Can the Themis+'s clues help diagnose tool weaknesses? Informed by the Themis+'s clues, the tool developers have successfully located several tool weaknesses, which were unknown or unclear before. We illustrate some found major tool weaknesses.

(1) Weaknesses in the event generation strategy. Most GUI testing tools parse GUI layouts to generate events. Specifically, they check the properties (*e.g.*, `clickable`, `long-clickable`) of the UI widgets to generate the UI events (*e.g.*, `click`, `long-click`). Themis+'s clues helped reveal some weaknesses in the event generation strategies of Fastbot and DroidBot, which degrade their bug finding abilities. For example, Figure 4 shows a `ListView` page (simplified from a bug in our study) and its GUI layout. In this layout, `ListView` is the root node and "A", "B" and "C" are the leaf nodes of `TextView` (wrapped by `LinearLayout`). From this layout, a "good" testing tool should generate three `click` events for "A", "B" and "C", respectively. However, for this case, WCTester and Ape succeed, but Fastbot and DroidBot fail. Because Fastbot generates an event only when a widget's `clickable` and `enabled` are both `True`, while DroidBot will not generate events for the nodes (*i.e.*, "A", "B" and "C") if the `clickable` property of their parent node (*i.e.*, `ListView`) is `True` [18]. As a result, Fastbot and DroidBot can only generate a `click` on `ListView` itself. WCTester and Ape succeed because they rewrite the `clickable` property of a leaf node (*i.e.*, "A", "B" and "C") by that of its parent node (*i.e.*, `ListView`) when the parent node is `clickable` [2]. Informed by EC, Fastbot's developer located this weakness and fixed its strategy by following Ape's.

(2) Weaknesses in the event selection strategy. Most testing tools select events for execution by some heuristic strategy. Themis+'s clues helped reveal some design issues in the event selection, which affect the bug finding abilities. For example, Fastbot implements a clustering strategy to group similar widgets to reduce search space. However, as we illustrated in Section 2.2, this strategy may unexpectedly decrease the probability of executing the events in the group. Fastbot was affected by this strategy on 3 bugs ("SN-114" is one of them). Informed by EPC, MD and ET (execution times of events), Fastbot has fixed this strategy with careful design. Additionally, DDroid's clues reveal that on the 8 out of 47 bugs, some testing tools can cover all the pivot events of the bug-triggering traces (*i.e.*, achieving 100% EC) but still miss these bugs. Informed by EPC and MD, we find that these tools fail to execute the pivot events in the right order. For such tool weaknesses, some tool developers plan to incorporate lightweight program analysis to improve the diversity of event selection.

(3) Other tool weaknesses. Based on Themis+'s clues, tool developers also found other tool weaknesses, including (1) failing to emulate the "search" event on the system keyboard or generate specific texts, (2) failing to interacting with external apps (*e.g.*, `Camera`, `File Chooser`, `Setting`), and (3) failing to support specific types of widgets or events (*e.g.*, `rotation` and `multi-touch`).

Table 5: Optimization results of WCTester and Fastbot.

Tool	#Missed	#Actionable	#Found	#Improved
WCTester	23	13	9	3
Fastbot	21	12	6	4

Can Themis^{+}'s clues improve the testing tools? *All* the tool developers explicitly stated that they would make tool enhancement based on the provided clues. Specifically, the developers of two actively-developing industrial testing tools, WCTester and Fastbot, have already made several improvements. Table 5 shows the enhancement results of the two optimized tools. Column "#Missed" is the number of bugs missed by the original tools, and "#Actionable" is the number of bugs for which the tool developers have devised actionable optimizations. "#Found" is the number of newly found bugs among "#Actionable", and "#Improved" is the number of bugs which are still missed but their coverage values have been improved. Note that not all the missed bugs could lead to actionable optimizations (see "#Missed" and "#Actionable") because some found tool weaknesses (*e.g.*, failing to cover the pivot events in the right order) are the open challenges [54]. In Table 5, we can see that WCTester and Fastbot have newly found 9 and 6 bugs respectively, and have improved the chance of finding 3 and 4 bugs in terms of the three coverage metrics respectively. It is clear that DDroid's clues have indeed helped improve these two tools. Note that the newly added optimizations are designed by developers in the general sense rather than overfitting specific bugs. We follow the same evaluation setup in RQ1 to assess the optimized tools.

How are the feedback of tool developers? We conducted a semi-structured interview [23, 31] with each of the seven tool developers. During the interviews, we solicited their feedback on the usefulness and usability of Themis^{+}'s clues. To sum up, *all* the developers give high rates on Themis^{+}'s clues, and appreciate that the visualized clues are intuitive for inspection. In particular, DroidBot's developer commented "*I usually use DroidBot's recorded UI trace graph to debug my tool, but it is very time-consuming for lengthy traces. Themis^{+}'s clues are exactly what I want.*" Fastbot's developer commented "*Themis^{+}'s* MD *metric is very useful. I can quickly know which events or screen pages I should focus on [for diagnosing]. It can save me a lot of time.*" WCTester's developer commented "*I routinely improve my testing tool by adding new code. But it is difficult to know how the new tool version works. Themis^{+} is nice as it can be used as a regression suite. That's very useful.*" Ape's developer commented "*Due to flakiness, replaying the recorded event trace [for debugging] is very difficult. I usually cannot find useful clues by manual analysis. Themis^{+}'s clues helped me a lot.*"

4.5 Results of RQ3

From **RQ2**, we know that the Themis^{+}'s clues are precise because no clues are contradictory with the manual analysis results of tool developers (see Table 4). Thus, we used the clues computed by our automata-based trace analysis approach as the ground truth, and validated the precision of *simple TC* and *simple RV* in identifying the first missed event in the bug-triggering trace.

Table 6 gives the overall evaluation results (the detailed results are provided in the supplementary material [15]). Column "Tools" lists the nine testing tools in our experiments. Column "#Cases" gives the total number of bugs missed by these tools according to the results of **RQ1**. Column "#Simple TC" and "#Simple RV" give the numbers of missed bugs for which simple TC and simple RV report the correct clues (which are consistent with the results of our approach, denoted by Column "#Themis^{+}"), respectively. Table 6 shows that simple TC and simple RV achieve low precision in finding correct clues. The precision of simple TC and simple RV ranges from 23.1~55.6% (computed by #simple TC/#Cases and #simple RV/#Cases per tool). For example, when analyzing the 23 bugs missed by WCTester, simple TC and simple RV find correct clues for only 8 and 9 missed bugs, respectively, achieving 39.1% (9/23) and 43.5% (10/23) precision, respectively. We can see that simple TC and simple RV are error-prone and unreliable. It indicates that our approach is really needed and the three automata-based metrics are useful.

Table 6: The number of correct clues on the first missed event reported by simple TC, simple RV and our approach.

Tool	#Simple TC	#Simple RV	#Themis^{+}	#Cases
WCTester	9	10	23	23
Fastbot	8	8	21	21
Ape	7	6	23	23
ComboDroid	11	7	25	25
Monkey	9	9	28	28
Stoat	14	15	27	27
DroidBot	10	13	28	28
Humanoid	6	6	26	26
Q-testing	12	13	24	24
#Total	86	87	312	312

4.6 Discussion

Precision and completeness of the automata. In our work, given all the minimal bug-triggering traces T_rs, the bug automaton is precise and complete by construction. Moreover, we empirically validated its precision: we randomly generate 100 random event traces from the automaton, and *all* the event traces reaching the final state indeed crashes the app. Thus, all the automata are precise. On the other hand, if a bug automaton is complete, the MD should be "0" when a tool triggers the bug (*i.e.*, the logged event trace when the crash happens should be accepted by the automaton). In Table 3, the symbol "*" on the values of MD denotes that the bug was triggered by a tool at runtime. We can see that, among the 9×47×5=2,115 tests (running 9 tools against 47 automata for 5 repeated runs), *only* 5 (≈0.2%) tests ("AB-375" for Monkey, and "AB-480" for WCTester, Fastbot, Ape and ComboDroid) fails the completeness. It indicates the tools may find some bug-triggering traces T_rs which were not reported in Themis (thus not included in the bug automata). When these traces are given, the automata could be complete. Thus, this is an orthogonal problem of our approach.

Manual *v.s.* automated automaton construction. In our work, the bug automata are manually constructed for Themis^{+}. It is similar to *manually* writing program specifications in formal verification [7, 24] (the bug automata can be viewed as the specification of undesired app behaviors). In Table 3, column "ϵ-NFA Sizes" gives the sizes of the automata. The minimal, median and maximum number of automaton states and transitions are 1, 5 and 17, *and* 1, 9 and 44, respectively. Thus, the complexity of bug automata is reasonable. In our experience, the construction effort ranges from

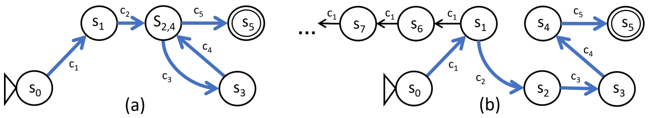

Figure 5: (a) imprecise automaton, (b) incomplete automaton

2∼20 minutes per automaton, which is acceptable. Note that the construction is *a one-time effort* — Themis+ is reusable for many different GUI testing tools. Thus, the benefits outweigh the effort.

Although many automated algorithms exist in building finite state machine based GUI models [1, 6, 11, 27, 53, 60], they are difficult to apply in our setting. Because defining one "apply-for-all" *state abstraction* criterion that fits all different apps is challenging [6]. As a result, these algorithms are difficult to guarantee the automaton's precision and completeness, which affects finding the right clues. Let us take `ScarletNotes`'s bug in Figure 1 as an example. Some state abstraction criteria (*e.g.*, C-Lv3, C-Lv4, C-Lv5 defined in [6]) abstracts l_2 and l_4 into the same state ($S_{2,4}$) because the UI layouts of l_2 and l_4 are identical. Figure 5(a) shows the partial automaton under such criteria. The automaton is imprecise because the trace $[c_1, c_2, c_5]$ reaching the final state is not a bug-triggering trace. On the other hand, we know that c_1 can be executed on l_1 to create new notebooks. If the state abstraction is sensitive to the number of created notebooks (*e.g.*, C-Lv4, C-Lv5 defined in [6]), a number of (possibly infinite) new states (*e.g.*, s_6, s_7) will be included into the automaton shown in Figure 5(b). It leads to an incomplete automaton. Additionally, the incompleteness could also be caused by inadequate explorations of different bug-triggering traces. As a result, we need extra manual efforts to validate (and fix) the automaton built by these algorithms.

Coverage metrics. The three coverage metrics `EC`, `EPC` and `MD` complement each other in finding the clues. No one is the best. For example, when `MD` is 0 (*i.e.*, the bug is triggered), `EC` and `EPC` may not reach 100%. Because a bug may have multiple bug-triggering traces, and the tool may only cover one trace. In this case, EC or `EPC` complements `MD` in understanding tool effectiveness. On the other hand, a tool may achieve 100% `EC` or `EPC` but may not achieve `MD` as 0. Because the tool covers all the events but fails to cover them in the right order. In this case, `MD` complements `EC` or `EPC`.

Threats to Validity. The first threat is the representativeness of the bugs in Themis. We emphasize that Themis's bugs are *diverse* (collected from 20 different apps), *nontrivial* (many bugs have long and complicated bug-triggering traces) and *selected without explicit bias* (only selecting critical bugs labelled by app developers). In the future, we would consider non-crashing bugs [56, 65]. The second threat is that our study involves human factors, *e.g.*, manually construct bug automata and letting the tool developers validate the clues from DDroid. To counter this, we empirically validated the precision and completeness of bug automata; and the developers are required to follow our instructions to carefully validate the clues to mitigate possible biases and we cross-checked the results.

5 RELATED WORK

Analyzing GUI testing tools for Android. To our knowledge, little prior work exists in analyzing tool weaknesses based on tool-missed bugs. For example, some work only compares different testing tools [12, 61] or evaluates specific testing strategies [3, 47, 50, 59] in terms of the achieved app code coverage and the number of found app crashes. They *do not* analyze potential tool weaknesses. Some work [8, 28, 67] *manually* inspect the uncovered app code to analyze the tool weaknesses of failing to achieve high app code coverage. Vet [62] uses two heuristic UI trace patterns to find the tool weaknesses in the form of UI exploration tarpits (*i.e.*, a tool is trapped for an excessive amount of time within a small fraction of app functionalities). However, these work in general *cannot* help analyze tool-missed bugs. For example, they can hardly help diagnose Fastbot against the bug in Figure 1. Because the bug does not have specific patterns of missed app code or UI exploration tarpits. Themis [54] is the only close work. But it can only *manually* analyze tool-missed bugs to understand tool weaknesses. Our work improves Themis by overcoming the difficulties of manual analysis.

Runtime verification and automata-based trace analysis. Runtime verification (RV) can help find (un)desired behaviors of the system under test [7]. The typical realization of RV is using a monitor (*e.g.*, an automaton synthesized from some system specification) to analyze the system's execution trace [24, 48]. For example, some work adapts the idea of RV to analyze system kernel traces [40] or debug specification violations [32]. At the high-level, our approach can be also viewed as the adaption of RV as the bug automaton is one form of program specifications. However, applying existing RV techniques for Android [13, 20, 57] in our setting is difficult. Because existing RV techniques focus on verifying generic (app-agnostic) properties (*e.g.*, good programming practices and security policies), which are manually described in temporal logic in terms of specific program APIs [44]. However, we concern app-specific bugs (involving diverse set of APIs), which are difficult to be captured by generic properties (and thus difficult to *automatically* synthesize the monitor like the bug automaton in our approach). The tools of these relevant work [13, 20, 57] are not available for comparison. AVA [5] uses a finite state automaton to represent the successful executions of a target system and use this machine to analyze the failing executions. AVA uses the deviated events from the failing executions to interpret why the system fails. Different from AVA, our approach uses the three different coverage metrics on the automaton itself to interpret why a target bug is missed.

6 CONCLUSION

In this paper, we introduce an automata-based trace analysis approach to tackling the challenge of manual trace analysis. Our approach can improve the classic benchmarking by providing the clues of tool weaknesses on the missed bugs. The evaluation confirms the feasibility and usefulness of our approach. Our work opens up a new perspective of analyzing the weaknesses of testing tools.

ACKNOWLEDGMENTS

We thank the anonymous ESEC/FSE reviewers for their comments and the testing tools' authors from ByteDance's SE lab (Zhao Zhang, Chao Peng, Tianxiao Gu), Tencent's WeChat team (Haochuan Lu, Ting Xiong, Yuetang Deng), and DroidBot/Humanoid (Yuanchun Li) for the discussions/feedback on our work. This work was supported in part by National Key Research and Development Program (2022YFB3104002, 2020AAA0107800), NSFC Grant 62072178, and Shanghai Trusted Industry Internet Software Innovation Center.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59. https://doi.org/10.1109/MS.2014.55

[2] Ape. 2022. *Ape's event generation strategy.* Retrieved 2022-3 from https://github.com/tianxiaogu/ape/blob/master/src/com/android/commands/monkey/ape/tree/GUITreeBuilder.java#L261

[3] Iván Arcuschin, Juan Pablo Galeotti, and Diego Garbervetsky. 2021. An Empirical Study on How Sapienz Achieves Coverage and Crash Detection. *Journal of Software: Evolution and Process* (2021), e2411. https://doi.org/10.1002/smr.2411

[4] ASM team. 2023. *ASM: an all purpose Java bytecode manipulation and analysis framework.* Retrieved 2023-1 from https://asm.ow2.io/

[5] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. 2009. AVA: Automated Interpretation of Dynamically Detected Anomalies. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/1572272.1572300

[6] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–249. https://doi.org/10.1145/2970276.2970313

[7] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to runtime verification. In *Lectures on Runtime Verification*. 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[8] Farnaz Behrang and Alessandro Orso. 2020. Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for Android. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1066–1077. https://doi.org/10.1145/3324884.3416567

[9] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*.

[10] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System. In *IEEE/ACM 1st International Conference on Automation of Software Test (AST)*. 93–96. https://doi.org/10.1145/3387903.3389308

[11] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 623–640. https://doi.org/10.1145/2509136.2509552

[12] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 429–440. https://doi.org/10.1109/ASE.2015.89

[13] Philip Daian, Yliès Falcone, Patrick O'Neil Meredith, Traian-Florin Serbanuta, Shin'ichi Shiriashi, Akihito Iwai, and Grigore Rosu. 2015. RV-Android: Efficient Parametric Android Runtime Verification, a Brief Tutorial. In *6th International Conference on Runtime Verification (RV) (Lecture Notes in Computer Science, Vol. 9333)*. 342–357. https://doi.org/10.1007/978-3-319-23820-3_24

[14] DDroid. 2022. *Themis+'s clues.* Retrieved 2023-1 from https://github.com/DDroid-Android/home/blob/main/README.md#html-report

[15] DDroid. 2023. *Supplementary materials for RQ2 and RQ3.* Retrieved 2023-8 from https://github.com/DDroid-Android/home/blob/main/supplementary-material.pdf

[16] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

[17] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 481–492. https://doi.org/10.1145/3377811.3380402

[18] DroidBot. 2022. *DroidBot's event generation strategy.* Retrieved 2022-3 from https://github.com/honeynet/droidbot/blob/master/droidbot/device_state.py#L401

[19] Caleb Evans. 2021. *automata-lib(5.0.0).* Retrieved 2021-12 from https://pypi.org/project/automata-lib/

[20] Yliès Falcone, Sebastian Currea, and Mohamad Jaber. 2012. Runtime Verification and Enforcement for Android Applications with RV-Droid. In *Third International Conference on Runtime Verification (RV) (Lecture Notes in Computer Science, Vol. 7687)*. 88–95. https://doi.org/10.1007/978-3-642-35632-2_11

[21] Fastbot team. 2022. *Fastbot(2.0).* Retrieved 2022-1 from https://github.com/bytedance/Fastbot_Android

[22] gcov team. 2023. *gcov-a Test Coverage Program.* Retrieved 2023-1 from https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[23] Tegan George. 2022. *Semi-Structured Interview: Definition, Guide and Examples.* Retrieved 2023-1 from https://www.scribbr.com/methodology/semi-structured-interview/

[24] Dimitra Giannakopoulou and Klaus Havelund. 2001. Automata-Based Verification of Temporal Properties on Running Programs. In *16th IEEE International Conference on Automated Software Engineering (ASE)*. 412–416. https://doi.org/10.1109/ASE.2001.989841

[25] Google Android team. 2023. *Transform.* Retrieved 2023-1 from https://developer.android.com/reference/tools/gradle-api/7.0/com/android/build/api/transform/Transform

[26] Google Inc. 2022. *Monkey.* Retrieved 2022-1 from https://developer.android.com/studio/test/monkey

[27] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications via Model Abstraction and Refinement. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 269–280. https://doi.org/10.1109/ICSE.2019.00042

[28] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. 2020. Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 557–568. https://doi.org/10.1109/ICSME46990.2020.00059

[29] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3 (2020), 49:1–49:29. https://doi.org/10.1145/3428334

[30] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. *Introduction to automata theory, languages, and computation, 2nd Edition.*

[31] Siw Elisabeth Hove and Bente Anda. 2005. Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research. In *11th IEEE International Symposium on Software Metrics (METRICS)*. 23. https://doi.org/10.1109/METRICS.2005.24

[32] Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, and Kevin Pouget. 2017. Interactive Runtime Verification - When Interactive Debugging Meets Runtime Verification. In *28th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 182–193. https://doi.org/10.1109/ISSRE.2017.19

[33] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*. 437–440. https://doi.org/10.1145/2610384.2628055

[34] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[35] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Trans. Reliability* 68, 1 (2019), 45–66. https://doi.org/10.1109/TR.2018.2865733

[36] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[37] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1070–1073. https://doi.org/10.1109/ASE.2019.00104

[38] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 224–234. https://doi.org/10.1145/2491411.2491450

[39] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 94–105. https://doi.org/10.1145/2931037.2931054

[40] Gabriel Matni and Michel R. Dagenais. 2009. Automata-based approach for kernel trace analysis. In *Proceedings of the 22nd Canadian Conference on Electrical and Computer Engineering (CCECE)*. 970–973. https://doi.org/10.1109/CCECE.2009.5090273

[41] Maubis. 2019. *Scarlet Notes.* Retrieved 2019-12 from https://play.google.com/store/apps/details?id=com.bijoysingh.quicknote

[42] Maubis. 2019. *Scarlet Notes's issue 114.* Retrieved 2019-2 from https://github.com/BijoySingh/Scarlet-Notes/issues/114

[43] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 2001. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE).* 256–267. https://doi.org/10.1145/503209.503244

[44] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. 2012. An overview of the MOP runtime verification framework. *Int. J. Softw. Tools Technol. Transf.* 14, 3 (2012), 249–289. https://doi.org/10.1007/s10009-011-0198-6

[45] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 153–164. https://doi.org/10.1145/3395363.3397354

[46] Jay Patel. 2018. *JFlap(7.1).* Retrieved 2021-10 from https://www.jflap.org/

[47] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the effectiveness of random testing for Android: or how I learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test (AST).* 34–37. https://doi.org/10.1145/3194733.3194742

[48] Giles Reger. 2014. *Automata based monitoring and mining of execution traces.* Ph. D. Dissertation. University of Manchester, UK. http://www.manchester.ac.uk/escholar/uk-ac-man-scw:225931

[49] Jakub Riegel. 2018. *PUTflap(1.0).* Retrieved 2021-12 from https://github.com/jakubriegel/PUTflap

[50] Leon Sell, Michael Auer, Christoph Frädrich, Michael Gruber, Philemon Werli, and Gordon Fraser. 2019. An Empirical Evaluation of Search Algorithms for App Testing. In *International Conference on Testing Software and Systems (ICTSS)*, Vol. 11812. 123–139. https://doi.org/10.1007/978-3-030-31280-0_8

[51] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: beyond GUI testing for Android applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 27–37. https://doi.org/10.1109/ASE.2017.8115615

[52] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why My App Crashes Understanding and Benchmarking Framework-specific Exceptions of Android apps. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3013438

[53] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* 245–256. https://doi.org/10.1145/3106237.3106298

[54] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* 119–130. https://doi.org/10.1145/3468264.3468620

[55] Ting Su, Jue Wang, and Zhendong Su. 2021. *The Themis Benchmark.* Retrieved 2022-3 from https://github.com/the-themis-benchmarks/home

[56] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages (OOPSLA)* (2021), 1–31. https://doi.org/10.1145/3485533

[57] Haiyang Sun, Andrea Rosà, Omar Javed, and Walter Binder. 2017. ADRENALIN-RV: Android Runtime Verification Using Load-Time Weaving. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).* 532–539. https://doi.org/10.1109/ICST.2017.61

[58] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal* 27, 1 (2019), 149–201. https://doi.org/10.1007/s11219-018-9418-6

[59] Thomas Vogel, Chinh Tran, and Lars Grunske. 2021. A comprehensive empirical evaluation of generating test suites for mobile applications with diversity. *Inf. Softw. Technol.* 130 (2021), 106436. https://doi.org/10.1016/j.infsof.2020.106436

[60] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE).* 469–480. https://doi.org/10.1145/3377811.3380382

[61] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE).* 738–748. https://doi.org/10.1145/3238147.3240465

[62] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: identifying and avoiding UI exploration tarpits. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* 83–94. https://doi.org/10.1145/3468264.3468554

[63] Tyler Wendland, Jingyang Sun, Junayed Mahmud, S. M. Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A Dataset of Manually-Reproduced Bug Reports for Android apps. In *18th IEEE/ACM International Conference on Mining Software Repositories (MSR).* IEEE, 600–604. https://doi.org/10.1109/MSR52588.2021.00082

[64] Wikipedia. 2022. *Floyd–Warshall algorithm.* Retrieved 2022-2 from https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm

[65] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 1319–1331. https://doi.org/10.1145/3597926.3598138

[66] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: are we really there yet in an industrial case?. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* 987–992. https://doi.org/10.1145/2950290.2983958

[67] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* 253–262. https://doi.org/10.1109/ICSE-SEIP.2017.32

[68] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (1997), 366–427. https://doi.org/10.1145/267580.267590

Received 2023-02-02; accepted 2023-07-27