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

Enze Ma¹, Shan Huang¹, Weigang He¹, Ting Su¹
Jue Wang², Huiyu Liu¹, Geguang Pu¹, Zhendong Su³
¹Shanghai Key Lab of Trustworthy Computing, Software Engineering Institute, East China Normal University, China
²State Key Lab for Novel Software Tech. and Dept. of Computer Sci. and Tech., Nanjing University, China
³Department 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.

KEYWORDS
Android GUI Testing, Runtime Verification, Trace Analysis

ACM Reference Format:

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 input events (steps): (1) c₁: clicking the notebook creation button (located at the bottom right on page l₀) to create a notebook (e.g., named as “Notebook1”); (2) c₂: clicking the created notebook “Notebook1” on page l₁ to enter into its directory; (3) c₃: opening the menu by clicking the menu button (located at the bottom left on page l₂); (4) c₄: choosing the “Locked” option on the menu to show the locked notes (page l₃); and (5) c₅: clicking the “x” button on page l₄ 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 “x” button to exit from that directory — it does

¹Enze Ma and Shan Huang contributed equally to this work, and they are co-first authors. Ting Su is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9327-0/23/12...
https://doi.org/10.1145/3611643.3616361

³A 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.
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., c1, c2, c3, c4, c5) is executable: the developer navigates the app to specific screen pages (i.e., l0, l1, l2, l3, l4), dumps the UI layouts, and checks whether the UI widgets (corresponding to c1, c2, c3, c4 and c5) 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 l0, l1, l2, l3 and l4, and exercise c1, c2, c3, c4 and c5 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
tool developers systematically, 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 a 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 AFE [27], ComboDroid [60], 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 3.3), 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, and $s_4$ 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 “X” 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 $c_5$) and $c_5$ (denoted by $c_5$). This transition helps capture such (non-minimal) traces as [$c_1$, $c_2$, $c_3$, $c_5$, $c_4$, $c_5$, $c_5$]. Additionally, one can press Back on page $l_2$, $l_3$ or $l_4$ to jump back to $l_1$, $l_2$, and $l_3$, respectively (denoted by the curved black lines in Figure 1(a)). As a result, one can take some other non-minimal traces, e.g., [$c_1$, $c_2$, $Back$, $c_3$, $c_4$, $c_5$, $c_6$] or [$c_1$, $c_2$, $c_3$, $c_4$, $Back$, $c_2$, $c_3$, $c_4$, $c_5$] to trigger the bug. To capture such traces, the bug automaton also includes these transitions enabled by Back, i.e., the transitions (denoted by $e$) from $s_2$ to $s_1$, $s_3$ to $s_2$, and $s_4$ to $s_3$. Specifically, $e$ 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 tool.

<table>
<thead>
<tr>
<th>Clue</th>
<th>WCTester</th>
<th>Fastbot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clue I: Event Coverage (EC)</td>
<td>2/5 (40.0%)</td>
<td>5/5 (100%)</td>
</tr>
<tr>
<td>Clue II: Event-Pair Coverage</td>
<td>4/24 (16.7%)</td>
<td>17/24 (70.8%)</td>
</tr>
<tr>
<td>Details of EC</td>
<td>(c_1) = (27), (c_2) = (35), (c_3) = (6), (c_4) = (0), (c_5) = (0), (c_6) = (1)</td>
<td>(c_1) = (81), (c_2) = (107), (c_3) = (17), (c_4) = (6), (c_5) = (6)</td>
</tr>
<tr>
<td>Details of EPC</td>
<td>(c_1), (c_2) = (0), (c_3), (c_4) = (0), (c_5), (c_6) = (0), (c_7), (c_8) = (11)</td>
<td>(c_1), (c_2) = (0), (c_3), (c_4) = (0), (c_5), (c_6) = (0)</td>
</tr>
<tr>
<td>Details of MD</td>
<td>([c_1, c_2]) is covered</td>
<td>([c_1, c_2, c_3, c_4]) is covered</td>
</tr>
</tbody>
</table>

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, \((c_1, c_2), (c_2, c_3), (c_5, c_6)\) 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.

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 diagnosis. 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 COMBDROID 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 “x” 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., \(w_3\sim w_4\)) are selected at the same level. As a result, the probability of executing \(c_5\) on \(l_4\) is \(\frac{1}{6}\) 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 \(e.t\) 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 \(T = [e_1, \ldots, e_n]\), where \(e_1\) is an event. When \(T\) is executed on an app \(P\), we can obtain a sequence of GUI layouts \(L\), i.e., \(L = [L_0, \ldots, L_{t-1}, L_t, \ldots, L_n]\), where \(L_0\) is the layout of the app starting page, and \(L_t\) is the layout due to the execution of \(e_t\) on \(L_{t-1}\) (\(1 \leq t \leq n\)). Intuitively, the execution of an event trace \(T\) can be represented as: \(L_0 \xrightarrow{e_0} L_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{t-1}} L_t \xrightarrow{e_t} L_{t+1} \ldots \xrightarrow{e_{n}} L_n\).

The main goal of an automated GUI testing tool \(\Gamma\) 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 \(T_r = [e'_1, \ldots, e'_j, \ldots, e'_m]\) (\(e'_j\) is an event), and the corresponding GUI layouts of \(T_r\) as \(L_r = [L'_0, \ldots, L'_{t-1}, L'_t, \ldots, L'_n]\) (\(L'_t\) 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,
Given a bug-triggering trace \( r \), we can formulate an initial bug automaton \( M \) in the form of a nondeterministic finite automaton \((\varepsilon\text{-NFA for short})[30]\).

**Definition 3.4. Bug Automaton.** A bug \( r \)'s automaton \( M \) is formulated as an \( \varepsilon \)-NFA. Given \( r \)’s minimal bug-triggering trace \( T_r = [e_1', \ldots, e_r', e_r'' \ldots, e_m'] \) and its corresponding GUI layouts \( L_r = \{e_0', \ldots, e_{r-1}', e_r', \ldots, e_m'\} \), we define \( M \) as \( (S, \Sigma, \delta, \delta, \varepsilon, S_0, F) \), where:
- \( \Sigma \) is a finite set of input symbols, \( \varepsilon \) included.
- \( \delta \) is a transition function, \( \delta: S \times \Sigma \rightarrow P(S) \), where \( P(S) \) 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 \( \Sigma \), the set of abstract program states \( S \), and the transition function \( \delta \) of the automaton \( M \). Specifically, the GUI layouts \( L_r \) are abstracted to the set of states \( S \), \( e_i' \) is abstracted to \( s_i \). Here, \( e_i' \) (the app’s starting page) is abstracted to \( s_0 \) (\( M \)'s initial state), and \( e_r'' \) (the app’s crashing page) is abstracted to \( s_m \) (\( M \)'s final state). According to the execution of \( T_r \), if there exists a page transition \( \ell_i \xrightarrow{c} e_i+1 \), the corresponding state transition \( s_i \xrightarrow{\ell_i} s_{i+1} \) will be added into the transition function \( \delta \). In this way, the initial bug automaton is constructed, i.e., \( s_0 \xrightarrow{c_1} s_1 \ldots s_{n-1} \xrightarrow{c_n} s_n \). Take ScarletNotes’s bug in Figure 1 as an 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 \( \delta \) 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 \( \Sigma \) can be executed on each state in \( S \) (except \( s_0 \) and \( s_5 \)) 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 \( \Sigma \) 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 \( \Sigma \) 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 \( \varepsilon \)-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 \( \varepsilon \) (at this time \( \Sigma \) is updated to \( \{c_1, c_2, c_3, c_4, c_5, \varepsilon\} \)) and include the transitions enabled by \( \varepsilon \). 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 \( l_1, l_2, l_1 \) respectively (denoted by the curved black lines in Figure 1(a)). Thus, we add the \( \varepsilon \)-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 \( [c_1, c_2, Back, c_2, c_3, c_4, c_5], [c_1, c_2, c_3, Back, c_2, c_3, c_4, c_5] \). After this step, we obtain the final automaton shown in Figure 3(c).
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 close to 1-minimal. (3) Given all the bug-triggering traces $T_s$, 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 $\epsilon$-NFA to its equivalent deterministic finite automaton (DFA) by eliminating the $\epsilon$ transitions. Formally, the DFA $M_d$ is $M_d = (S_d, \Sigma_d, \delta_d, q_0, F_d)$, where all the components have their similar interpretations as for the $\epsilon$-NFA and $\Sigma_d = \Sigma \setminus \{ \epsilon \}$. We conduct this conversion because the DFA (without $\epsilon$-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 $\Sigma$) against $M_d$. (2) The conversion is not expensive as the sizes of $\epsilon$-NFAs are relatively small. Moreover, $\epsilon$-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, "$\epsilon$-NFA Sizes" and "DFA Sizes" show the sizes of $\epsilon$-NFA and its DFA, respectively.

### 3.3 Coverage Metrics based Clues

We introduce three coverage metrics at the automaton level of $M_d$ (which only contains the symbols in $\Sigma$). The $\epsilon$-NFA version is not expensive as the sizes of $M$ are significantly smaller.

#### Clue I: Event Coverage

Let $E_d$ be the set of all the events in $\Sigma_d$, and let $E_c$ be the set of events executed by a testing tool $\Gamma$. Formula (1) defines event coverage (EC) to characterize how many events could be covered by $\Gamma$.

$$EC = \frac{|E_c|}{|E_d|} \times 100\% \quad (1)$$

Conceptually, EC is similar to the statement coverage in classic software testing. Since the events in $\Sigma_d$ are necessary to trigger $r$, EC can assess the tool effectiveness when $\Gamma$ cannot execute all these events. The higher EC is, the more likely $\Gamma$ can find the bug $r$. If $\Gamma$ cannot execute some events, it likely 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 II: Event-Pair Coverage

Let $L_p$ be the set of all event pairs $(e_x, e_y)$ in $M_d$, where $e_x$ and $e_y$ are the events in $\Sigma_d$ and denote the events of two adjacent transitions in $\delta_d$. 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, $L_p = \{(e_x, e_y) | \exists s_i, s_j, s_k \in \Sigma_d, e_x, e_y \in \Sigma_d, \delta_d(s_i, e_x) = q_j \land \delta_d(s_j, e_y) = s_k \}$. Let $L_c$ 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 $\Gamma$.

$$EPC = \frac{|L_c|}{|L_p|} \times 100\% \quad (2)$$

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 $\Gamma$ to execute two adjacent transitions, and (2) reflect the diversity of event traces generated by $\Gamma$. The higher EPC is, the more likely $\Gamma$ 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 $T_{\Sigma_d} = \{ e_1, \ldots, e_j \}$ be the event trace generated by a testing tool $\Gamma$. Let $S_d = \{s_0, s_1, \ldots, s_m\}, s_j \in S_d, 1 \leq j \leq m$ be the set of states that $T_{\Sigma_d}$ can reach when matching $T_{\Sigma_d}$ 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 $\Gamma$ can reach a crash bug $r$.

$$MD = min(distance(s_j, F_d) | s_j \in S_d) \quad (3)$$

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

For example, if $S_d = \{s_0, s_1, s_2, s_3\}$ 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 $c_3, c_1,$ 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 tool $\Gamma$ can exercise the events of the bug-triggering trace in some specific orders, and (2) quantify how far $\Gamma$ is to reach $r$ in terms of number of events to be executed. It indicates the ability boundary of a testing tool. If $\Gamma$ 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 $L_p$ 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 $T_r = \{ e'_1, \ldots, e'_t, \ldots, e'_m \}$, we instrument $P$ at the event listener of each event $e'_t$. In this way, $e'_i$ will be logged when it is executed by the tool $\Gamma$.

2. **Logging.** The testing tool $\Gamma$ is run against the instrumented app $P$ to log the executed pivot events. All the logged pivot events forms an event trace $L$. $\Gamma$ 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 $\epsilon$-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,
We use automata-lib [19] to convert an RQ3: visualize the clues via interactive automata-based trace analysis for aiding diagnosing GUI testing tools for Android ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA

these selected tools and their main testing strategies. Readers can obtained on request from WeChat’s testing team. Table 2 summarizes these selected tools and the main testing strategies.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Venue/Source</th>
<th>Main Testing Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOAT</td>
<td>ESEC/FSE ’17</td>
<td>Model-based</td>
</tr>
<tr>
<td>DroidBot</td>
<td>ICSE ’17</td>
<td>Model-based</td>
</tr>
<tr>
<td>Ape</td>
<td>ICSE ’19</td>
<td>Model-based</td>
</tr>
<tr>
<td>Humanoid</td>
<td>ASE ’19</td>
<td>Deep learning-based</td>
</tr>
<tr>
<td>ComboDroid</td>
<td>ICSE ’20</td>
<td>Model-based</td>
</tr>
<tr>
<td>Q-testing</td>
<td>ISSTA ’20</td>
<td>Reinforcement learning-based</td>
</tr>
<tr>
<td>Monkey</td>
<td>Google</td>
<td>Random testing</td>
</tr>
<tr>
<td>Fastbot</td>
<td>ByteDance</td>
<td>Model- &amp; Reinforcement learning-based</td>
</tr>
<tr>
<td>WCTester</td>
<td>WeChat</td>
<td>Random &amp; Reinforcement learning-based</td>
</tr>
</tbody>
</table>

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 Grable 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’s provide the clues on the bugs missed by automated GUI testing tools, compared to Themis?
- **RQ2**: How useful are the clues provided by Themis’s 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 SAPIENT [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’s 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 automaton 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 $47 \times 6 \times 5 \times 9 = 12,690$ 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.
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🪤 of a known bug 𝑒. It reports the first differing event between these two traces 𝑇 and 𝑇🪤. 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 computed 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 𝑇̃ = [𝑒1, ⋯, 𝑒𝑖, ⋯, 𝑒𝑛], 𝑒𝑖 is the first missed event in 𝑇̃ if 𝑒𝑖 is missed but all the events 𝑒1, ⋯, 𝑒𝑖−1 are covered in the order by 𝑇̃. 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 vs. 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 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, FastB00T) 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
focus on these achieved best values as they indicate the best tool performance. Take the results of WC\textsc{Tester} on bug “SN-114” as an example (see row “SN-114” under column “WC\textsc{Tester}”), 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 WC\textsc{Tester}.

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\textsuperscript{*}'s clues? Table 4 gives the validation results on Themis\textsuperscript{*}'s clues. Column “\#Identical”, “\#Useful” and “\#Misleading” denote the numbers of missed bugs for which Themis\textsuperscript{*} 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\textsuperscript{*}'s clues are identical or useful compared to the clues manually found by developers, without any misleading ones. Specifically, Themis\textsuperscript{*} 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.

#### How can Themis\textsuperscript{*} find identical or useful clues? In 71 cases, Themis\textsuperscript{*} 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\textsuperscript{*} can find the identical clue as EC can tell the multi-touch event is not covered. In 102 cases, Themis\textsuperscript{*} 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, WC\textsc{Tester}'s developer cannot find any clue although he observes that the tool could click all the widgets of \(e_1\)~\(e_5\). Themis\textsuperscript{*} finds the clue that WC\textsc{Tester} 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\textsuperscript{*} reveals that WC\textsc{Tester} 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.](image)

Can the Themis\textsuperscript{*}'s clues help diagnose tool weaknesses? Informed by the Themis\textsuperscript{*}'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\textsuperscript{*}'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 List\textit{View} page (simplified from a bug in our study) and its GUI layout. In this layout, List\textit{View} is the root node and “A”, “B” and “C” are the leaf nodes of Text\textit{View} (wrapped by Linear\textit{Layout}). From this layout, a “good” testing tool should generate three click events for “A”, “B” and “C”, respectively. However, for this case, WC\textsc{Tester} 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., List\textit{View}) is true [18]. As a result, Fastbot and DroidBot can only generate a click on List\textit{View} itself. WC\textsc{Tester} 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., List\textit{View}) 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\textsuperscript{*}'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 EC, 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 EC 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\textsuperscript{*}'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).
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 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 DroidBot’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 3MDF 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’s clues are 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.

4.6 Discussion
Precision and completeness of the automata. In our work, given all the minimal bug-triggering traces $T_b$s, 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) fail the completeness. It indicates the tools may find some bug-triggering traces $T_b$s 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 vs. 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 “e-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...
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, Yueyang 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, 2020AA0107800), NSFC Grant 62072178, and Shanghai Trusted Industry Internet Software Innovation Center.
Automata-based Trace Analysis for Aiding Diagnosing GUI Testing Tools for Android

Giles Reger. 2014.