An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools

Wenyu Wang
University of Illinois at Urbana-Champaign, USA
wenyu2@illinois.edu

Wing Lam
University of Illinois at Urbana-Champaign, USA
winglam2@illinois.edu

Tao Xie*
Peking University, China
taoxie@pku.edu.cn

ABSTRACT

Due to the importance of Android app quality assurance, many Android UI testing tools have been developed by researchers over the years. However, recent studies show that these tools typically achieve low code coverage on popular industrial apps. In fact, given a reasonable amount of run time, most state-of-the-art tools cannot even outperform a simple tool, Monkey, on popular industrial apps with large codebases and sophisticated functionalities. Our motivating study finds that these tools perform two types of operations, UI Hierarchy Capturing (capturing information about the contents on the screen) and UI Event Execution (executing UI events, such as clicks), often inefficiently using UIAutomator, a component of the Android framework. In total, these two types of operations use on average 70% of the given test time.

Based on this finding, to improve the effectiveness of Android testing tools, we propose Toller, a tool consisting of infrastructure enhancements to the Android operating system. Toller injects itself into the same virtual machine as the app under test, giving Toller direct access to the app's runtime memory. Toller is thus able to directly (1) access UI data structures, and thus capture contents on the screen without the overhead of invoking the Android framework services or remote procedure calls (RPCs), and (2) invoke UI event handlers without needing to execute the UI events. Compared with the often-used UIAutomator, Toller reduces average time usage of UI Hierarchy Capturing and UI Event Execution operations by up to 97% and 95%, respectively. We integrate Toller with existing state-of-the-art/practice Android UI testing tools and achieve the range of 11.8% to 70.1% relative code coverage improvement on average. We also find that Toller-enhanced tools are able to trigger 1.4x to 3.6x distinct crashes compared with their original versions without Toller enhancement. These improvements are so substantial that they also change the relative competitiveness of the tools under empirical comparison. Our findings highlight the practicality of Toller as well as raising the community awareness of infrastructure support's significance beyond the community's existing heavy focus on algorithms.

*Tao Xie is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China, and is the corresponding author.

CCS CONCEPTS

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

KEYWORDS

UI testing, test generation, Android framework

ACM Reference Format:
Wenyu Wang, Wing Lam, and Tao Xie. 2021. An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460319.3464828

1 INTRODUCTION

As the Android operating system continues to thrive [25], effective quality assurance for Android apps has become increasingly demanded by Android app developers. The ever-growing complexity and fast pace of app feature development have imposed unprecedented challenges on making these Android apps robust and reliable. User Interface (UI), as the primary medium of user interactions, is naturally a good entry point for testing. UIs generally expose vast functionalities through unified interfaces, making them a good fit for automated testing. While manual and scripted UI testing is a common practice, automated UI testing is becoming increasingly popular. Automated testing complements manual testing by requiring little to no human testing effort. Developers and testers can easily run automated UI tests anytime, for long periods of time, and for multiple apps across many devices. Besides aiming to achieve comparable coverage of app functionalities with human efforts, such tests can also help with more thoroughly covering app logic that could be overlooked by human testers.

There have been numerous Android UI testing tools from both the research community and industry after years of development. *Monkey* [15], an Android UI testing tool developed by Google, is one of the earliest efforts in this direction. Shipped by default with most Android devices, the tool produces purely randomized UI event sequences and injects them into the target Android system without considering the design details of the app under test. Developed after the release of Monkey, various testing tools aim to improve Monkey's simple testing strategy. They are mainly randomness-driven/evolutionary-algorithm-based tools [29, 31, 39], model-based tools [6, 18, 21, 28, 37], and systematic-exploration-based tools [2, 4, 30]. In these tools' paper publications, the authors all claim that their tools can perform better than Monkey when the tools are given a reasonable amount of run time.

Unfortunately, subsequent work [7, 38] finds that most of the existing tools barely outperform Monkey (w.r.t. code coverage and crash triggering ability) when the tools are given a reasonable amount of run time. A study [7] in 2015 draws this conclusion based on experimental results of 10 tools (other than Monkey) on 68 relatively simple, open-source apps. A number of tools continue to be published after this work, and subsequently, another study [38] in 2018 reaffirms the conclusion based on experimental results of 5 tools (other than Monkey; 4 of them are tools not studied in [7]) on 68 popular and relatively complex, industrial apps from the Google Play store [13]. The conclusion from these two studies [7, 38] contradicts researchers' and practitioners' common belief: given that these tools have much more sophisticated design, they should all outperform Monkey.

To understand what makes existing tools perform poorly on popular industrial apps, we conduct a motivating study to find what types of operations from these tools use most of the run time, with particular attention given to testing infrastructure's impacts on testing effectiveness; these impacts have been constantly overlooked by prior work [2, 4, 6, 18, 21, 29–31, 37, 39], which mainly emphasizes algorithmic improvements. Our study findings show that capturing information about the contents on the screen (*UI Hierarchy Capturing*) and executing UI events (*UI Event Execution*) are the two types of operations that consume the most time. In total, these two factors use on average 70% of the entire run time budget with 34% and 36% belonging to UI Hierarchy Capturing and UI Event Execution, respectively (as shown in "Combined" in Figure 2). These two types of operations, usually provided with *infrastructure support*, are essential for most UI testing tools to perform their duties. Yet, we find that these two types of operations are often performed inefficiently using UIAutomator [22], a component of the Android framework. For example, we find that it can take from 0.4 to 8.2 seconds on average to capture one UI hierarchy using UIAutomator (as shown in Table 2). Our experiments (Section 5.2) find that these time usages can be reduced to just tens of milliseconds with infrastructure enhancements. The findings from our study suggest that there are substantial efficiency improvements that can be achieved with infrastructure enhancements so that tools can be more effective when given the same run time.

Based on the aforementioned findings, we propose TOLLER, a tool to provide infrastructure enhancements for UI Hierarchy Capturing and UI Event Execution to Android UI testing tools. By modifying the Android framework, TOLLER is capable of injecting itself into any target app's virtual machine and has access to the app's runtime memory. TOLLER can thus directly read an app's internal UI data structures and quickly extract the app's UI hierarchy, avoiding much of the overhead caused by using UIAutomator, which relies on the complicated internal logic of the Android framework as well as remote procedure calls. TOLLER also enables the direct invocation of UI event handlers, thereby eliminating the unnecessary time spent on executing low-level UI events that simulate human interactions (e.g., waiting for long clicks) and have to be translated to UI element-specific events based on the UI hierarchy. Our experiments show that TOLLER can substantially reduce the time required for the aforementioned two types of operations.

We integrate TOLLER with three state-of-the-art or state-of-the-practice Android UI testing tools that depend on UIAutomator: *Stoat* [37], *WCTester* [40, 41], and a tool named *Chimp*, which we implement following a similar algorithmic design as the original *Monkey* [15]. Our experiments with 15 popular, industrial apps obtained from the Google Play Store show that the average time usages for UI Hierarchy Capturing are reduced by about 97%, 77%, and 97% on Chimp, WCTester, and Stoat, respectively, and for UI Event Execution, the average time usages are reduced by 40%, 18%, and 95% on Chimp, WCTester, and Stoat, respectively.

Given the same run time for tools with and without TOLLER, the TOLLER-enhanced tools are able to execute more events than the original versions of the tools, and the TOLLER-enhanced tools on average achieve higher code coverage and trigger more distinct crashes than the original versions of the tools. In fact, our experiments show that TOLLER-enhanced tools achieve 11.8% to 70.1% relative code coverage improvement on average and are able to trigger 1.4x to 3.6x distinct crashes compared with the original versions of the tools. These improvements are so substantial that they even change the relative competitiveness of the tools under empirical comparison. For instance, Stoat achieves a higher average code coverage compared to Monkey only after Stoat is enhanced with TOLLER. We additionally involve another Android UI testing tool, Ape [18], in our experiments. We find that Ape is already benefiting from its own improved infrastructure support, despite the fact that the tool authors did not mention the improved infrastructure support in their paper. We make TOLLER's source code and the scripts used to set up TOLLER publicly available [1]. We hope that our results can raise the community's awareness of the significance of infrastructure support beyond the community's existing heavy focus on algorithms.

This paper makes the following main contributions:

- A motivating study to understand what types of operations use most of existing Android UI testing tools' run time. Our results show the potential of two infrastructure enhancements for improving these tools' testing efficiency.
- Design and implementation of TOLLER, which provides two infrastructure enhancements to Android UI testing tools so that they can benefit from efficient UI Hierarchy Capturing and UI Event Execution support.
- Comprehensive experiments involving the integration of TOLLER with Android UI testing tools. Our experiments show that infrastructure enhancements can lead to substantial effectiveness improvements.

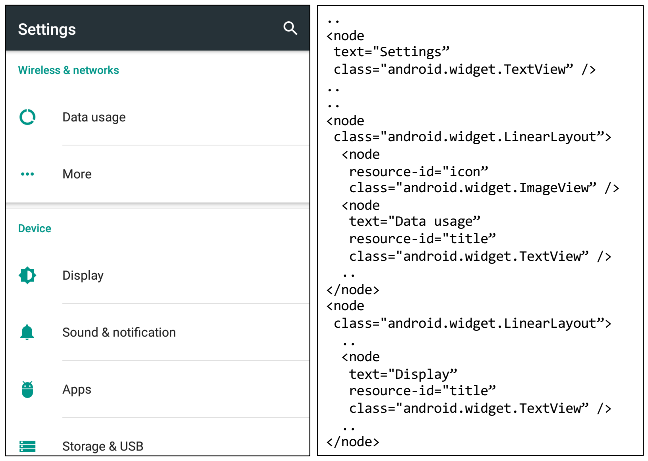
2 BACKGROUND

This section presents the relevant background information of TOLLER, including two Android UI system interfaces used mainly by testing tools, the structure of the Android framework, and UIAutomator.

2.1 Android System Interfaces for Testing Tools

This section introduces the two Android UI system interfaces that are used by most testing tools and that TOLLER aims to tackle: *UI Hierarchy Capturing* and *UI Event Execution*.

UI Hierarchy Capturing enables UI testing tools to obtain detailed information about current on-screen contents, including UI properties (e.g., widget type, location, and size) and hierarchical settings (e.g., some widget being a child of another widget). The captured UI information serves as context for testing decision making and is especially critical to model-based tools.



```
..
<node
 text="Settings”
 class="android.widget.TextView” />
..
..
<node
 class="android.widget.LinearLayout”>
  <node
   resource-id="icon”
   class="android.widget.ImageView” />
  <node
   text="Data usage”
   resource-id="title”
   class="android.widget.TextView” />
  ..
</node>
<node
 class="android.widget.LinearLayout”>
  ..
  <node
   text="Display”
   resource-id="title”
   class="android.widget.TextView” />
  ..
</node>
```

Figure 1: A simplified example of captured UI hierarchy

Figure 1 shows a simple example of a captured UI hierarchy (represented in XML format) along with its corresponding screenshot. Each `node` in the hierarchy depicts a `View` (abstraction of UI elements), which can be either a `ViewGroup` (`Views` specifically for holding and organizing other `Views`) or ordinary `View` (i.e., UI elements) that users can see and interact with. The hierarchical relations among `Views` are reflected by "child of" relations of `nodes`. Each `node` contains UI properties (e.g., text, position, resource ID) that vary across different types and instances of UI elements.

UI Event Execution enables tools to perform UI events (e.g., screen clicks, text inputs) on the app under test. The interface is usually invoked after each UI Hierarchy Capturing, where at each step, a tool gets the current UI state and then executes a UI event based on the state of the UI. A common way of performing UI events is to inject the corresponding *low-level UI events* into the Android system. For example, long clicking some point (x, y) on the screen can be decomposed into (1) touching down at (x, y), (2) waiting for 0.5 seconds, and (3) touching up at (x, y). These actions are processed as if they were from human users.

2.2 Structure of Android Framework

This section introduces how app bytecode runs on Android, as well as how the Android framework is structured. This section helps explain how Toller is integrated with the Android framework.

The Android framework can be divided into two parts: the *app space* part, which runs in the same virtual machine (VM) as each app's bytecode, and the *system service* part, which runs in standalone VMs and communicates with the app space part through RPCs. The app space part of the Android framework consists of a number of fundamental Java classes that are accessible from every Android app. These fundamental classes are preloaded into the VM and cannot be overridden by app classes. These characteristics of the app space part make it ideal to host Toller's runtime stub, which needs to gain direct access to each app's runtime memory. Section 4 presents more details about how Toller makes use of direct access to app runtime memory and the benefits of doing so.

2.3 UIAutomator

One component of the Android framework is UIAutomator [22], the standard service for UI interactions on an Android device, used by not only automated UI testing tools but also UI test scripting platforms such as Espresso [10].

Implementation of UIAutomator can be divided into three parts. The first part runs as a system service, which coordinates all UIAutomator related activities on the device. The second part resides in the app space Android framework, responsible for collecting UI-related information from the app runtime memory and communicating with the system service counterpart. The third part acts as a client to the system service, with which a user can request UI information to be captured or UI event to be executed.

There is much overhead in using UIAutomator, especially when it is used to capture UI hierarchies. When a user sends a request to the system service for UI Hierarchy Capturing through remote procedure calls (RPCs), the service first needs to look up the active UI windows and then dispatch the request using RPCs to each app process owning the UI windows. When the app space Android framework counterpart receives the request, it uses accessibility interfaces to gather the UI hierarchy for each requested UI window and transmits the UI hierarchies back to the system service. When each app process has finished processing, the system service finally formats the UI hierarchies into one XML document and then transmits the document back to the user. Section 4 presents details for how Toller can reduce this overhead.

3 MOTIVATING STUDY

A recent study [38] has found that Android testing tools are substantially less effective on popular industrial apps, compared with open-source apps, which are often used for evaluation. Many popular industrial apps are feature-rich and have much larger codebases than open-source apps. Existing work [2, 4, 6, 18, 21, 29–31, 37, 39] has been focusing on designing sophisticated UI exploration algorithms to achieve better testing effectiveness. Although testing effectiveness can be improved with sophisticated algorithms, one often overlooked aspect is the efficiency of infrastructure support, which is necessary for tools to perform their duties. Given that a common way of evaluating Android testing tools is to set a run time limit and measure code coverage or crash triggering ability at the end of the run time, the efficiency of infrastructure support directly affects a tool's overall testing effectiveness.

To guide enhancements to Android testing tools, we conduct a motivating study to understand the extent and sources of inefficiency from infrastructure support. Our study focuses on understanding the (in)efficiencies of Android UI testing tools interacting with the testing devices. Our findings enable us to design and implement a general solution for different tools. While UI Hierarchy Capturing and UI Event Execution are necessary parts of tool-device interactions, testing tools can also have other types of interactions. For example, a tool may execute a shell command through Android Debug Bridge (ADB) [14] to start the target app. Our motivating study aims to understand the time usages by different types of interactions to learn about their potentials for enhancements.

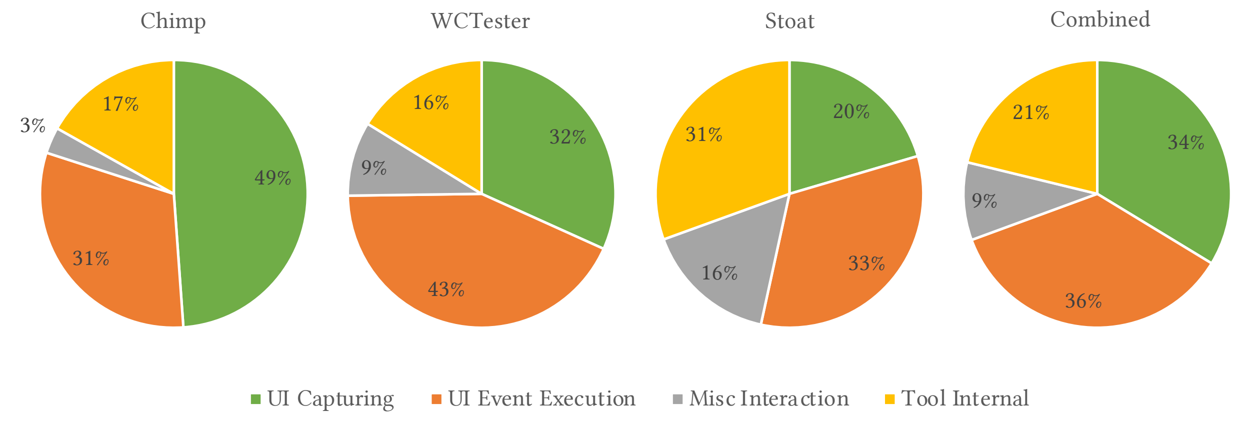


Figure 2: Time usage distribution by operation types

Table 1: Overview of industrial apps for experiments

App Name	Version	Category	#Inst	APK Size
Abs	4.2.0	Health & Fitness	10m+	57 MB
Duolingo	3.75.1	Education	100m+	12 MB
Filters For Selfie	1.0.0	Beauty	1m+	21 MB
GoodRx	5.3.6	Medical	1m+	12 MB
Google Translate	6.5.0.RC04	Tools	500m+	26 MB
Marvel Comics	3.10.3	Comics	5m+	6.2 MB
Merriam-Webster	4.1.2	Books & Reference	10m+	66 MB
Mirror	30	Beauty	1m+	3.3 MB
My baby Piano	2.22.2614	Parenting	5m+	3.7 MB
Sketch	8.0.A.0.2	Art & Design	50m+	25 MB
trivago	4.9.4	Travel & Local	10m+	12 MB
WEBTOON	2.4.3	Comics	10m+	23 MB
Word	16.0.9126	Productivity	100m+	74 MB
YouTube	15.35.42	Video Player & Editor	1b+	93 MB
Zedge	7.2.2	Personalization	100m+	33 MB

Notes: '#Inst' denotes the approximate number of downloads.

3.1 Experiment Settings

To drive our design of Toller, we run and profile three tools: Chimp, WCTester [41], and Stoat [37]. All three of these tools use UIAutomator [22] and control the testing devices from a computer (i.e., having no on-device components themselves), making it easy to profile the tools' interactions with the testing devices. The following are more details about these three tools:

- We implement *Chimp*, a tool based on Monkey [15]. Similar to Monkey, Chimp randomly decides on what UI events to generate. The main difference between Chimp and Monkey is that Chimp is aware of the UI element locations when generating UI events while Monkey is unaware. Note that we do not include Monkey directly in our study because it does not capture any UI information from the target app and just injects random low-level UI events. Therefore, the enhancements provided by Toller are unlikely to improve Monkey's testing effectiveness. On the other hand, Chimp shares the same exploration strategy as Monkey and makes use of Toller's infrastructure enhancements. Specifically, at each step, Chimp (1) obtains the current UI hierarchy, (2) determines what UI event types are executable (i.e., there is at least one UI element with a corresponding action handler) on the current screen, (3) randomly chooses a UI event type based on a predefined probability distribution, and (4) randomly chooses an applicable UI element (and action parameters if needed) to apply the next action on.
- *WCTester* [41] is a practical upgrade from Monkey, featuring widget awareness, state awareness, and various heuristics to improve its effectiveness. Developed by researchers and practitioners [40, 41], the tool has been deployed on WeChat, an app with over one billion monthly active users. The tool has moderate testing effectiveness on various industrial apps according to a previous study [38]. Although WCTester is not open-sourced, the authors [40, 41] shared the tool with us upon request.
- *Stoat* [37] is a sophisticated model-based tool featuring probabilistic modeling and sampling-based model evolution. While the Stoat paper [37] reports that Stoat outperforms Monkey based on an evaluation using open-source apps, a previous study [38] shows that Stoat generally achieves low code coverage on popular industrial apps and achieves lower code coverage than Monkey. Stoat is open-sourced.

All of our experiments are conducted on the official Android x86-64 emulators running Android 6.0 on a server with Xeon E5-2650 v4 processors. Each emulator is allocated with 4 dedicated CPU cores, 2 GiB of RAM, and 2 GiB of internal storage space. The emulators are stored on a RAM disk and backed by discrete graphics cards for minimal mutual influences caused by disk I/O bottlenecks and CPU-intensive graphical rendering.

Our experiments consist of 15 widely used industrial apps from the Google Play Store and are selected from obtaining the top apps from 13 different categories. The selected apps must all run on Android 6.0 x86-64 emulators and do not require logging in, given that logins can be flaky due to network calls and require expensive manual checks afterwards. Details of these apps are shown in Table 1. Each testing tool runs on each app for one hour without interruption. If a tool exits before using up the run time budget in some run, we automatically launch the tool to start testing again until the allotted one hour is up.

3.2 Results

Figure 2 shows the breakdown of time usages by different types of tool operations. Specifically, we measure the number of occurrences as well as the end-to-end time usages of three types of operations:

UI Hierarchy Capturing, UI Event Execution, and ADB command executions that are not used for the first two purposes (labeled as "Misc Interaction"). The remaining time used during testing is then considered to be used internally by the tool.

As shown in Figure 2, UI Hierarchy Capturing and UI Event Execution take most of the run time budget on all of the studied tools. By putting results from all three tools together ("Combined"), we see that these two types of interactions each take about 1/3 of the entire run time. Our findings suggest that focusing on the two types of operations has good potential for improving the efficiency of these tools and consequently the effectiveness of the tools when the tools are given a specific run time budget. In Section 5, we show how the use of Toller to enhance UI Hierarchy Capturing and UI Event Execution can lead to higher average code coverage and better crash triggering ability for the same three tools and 15 apps used in this motivating study.

4 DESIGN & IMPLEMENTATION OF TOLLER

Figure 3 shows an overview of Toller's design. Toller's source code and our scripts to set up Toller are publicly available [1]. Toller resides in the same VM as the app under test, giving Toller fast and direct access to the app's runtime memory. Toller is thus able to (1) resolve the app's internal UI-related data structures to generate UI hierarchies, and (2) dynamically analyze, invoke, or alter UI event handlers to perform UI events or understand/control app behaviors. To resolve UI-related data structures, Toller uses Java reflection to read the single-instanced `AccessibilityManager` class that indirectly points to all visible windows' root view groups. Toller then recursively finds all Java objects corresponding to child views (i.e., instances of subclasses of `android.view.View`) to generate hierarchies. To invoke UI event handlers, Toller directly calls the corresponding action invocation methods (e.g., `performClick()`) upon `View` objects. Note that Toller's UI Event Execution strategy falls back to low-level UI event injections on event handlers that have not been covered by a low-level UI event injection. This strategy helps Toller by (1) invoking low-level UI event injections at least once for every event handler that Toller directly invokes and (2) preventing the direct invocation of event handlers from covering less code than low-level UI event injections. These low-level injections can cover more code than directly invoking a specific event handler, say *EH*, because low-level injections may first invoke a topmost `View`'s event handler only for it to then invoke a child `View`'s event handler until the event eventually reaches *EH*.

We also design Toller to be non-intrusive to the app under test: Toller is bundled with the app-space Android framework classes on the testing device and app installation packages are not modified. This design is particularly useful for testing close-sourced industrial apps, because (1) many apps have self-protection mechanisms, preventing unauthorized changes to the installation packages, and (2) manipulating a large app's bytecode is highly error-prone (for example, just adding a new class during instrumentation may cause an app's `.dex` file to exceed the 64K method limit [12]).

For our experiments, we integrate Toller with the Android 6.0 framework on both emulators and real devices. While we experience no issue with our way of integration, we would still like to point out that it is possible to use Toller without modifying the

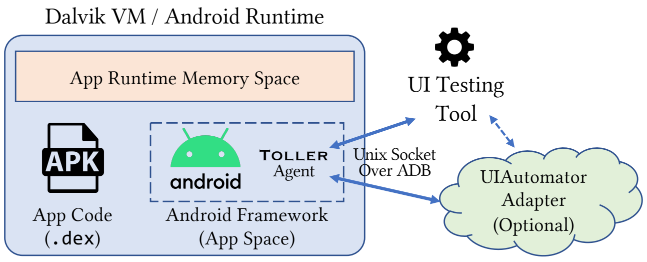


Figure 3: Overview of Toller's design

Android framework; in such a case, developers could simply need to add Toller to the app's codebase when building the app. We discuss more about the trade-offs of this option in Section 7. In brief, we take the following steps to inject Toller into an Android framework. First, we obtain the Android framework's DEX bytecode from the target device using ADB. Second, we convert the Android framework's bytecode into Smali [26] IR code. Third, we compile Toller's source code into Smali code. Fourth, we modify the Android framework's Smali code to incorporate Toller's Smali code. Finally, we convert all Smali code into DEX bytecode and replace the Android framework's bytecode on the target device with the converted DEX bytecode.

As Android testing tools typically run on a computer, while the app under test and Toller run on a device, the tools and Toller need a mechanism to communicate (e.g., share UI hierarchy information) with one another. Toller uses Unix's abstract socket for this communication as Android already provides good support (e.g., `LocalServerSocket`) for such communication. By doing so, Toller also does not need the app to have any specific permissions (e.g., networking, read/write storage).

As developers often change `UncaughtExceptionHandlers` when apps are first started to avoid sharing implementation details, Toller can also be used to disable apps' `UncaughtExceptionHandlers`. Disabling such handlers helps ensure that stack traces are printed to system logs so that experiments can understand crash statistics from such logs (e.g., our experiments in Section 5.4). To disable such handlers, Toller periodically (every five seconds in our experiments) calls `Thread.setDefaultUncaughtExceptionHandler()` to restore the default handler to ensure that stack traces from crashes are printed to the system logs.

Toller's implementation of UI Event Execution relies on Toller's UI Hierarchy Capturing. Concretely, in a Toller-captured UI hierarchy, each UI element is associated with a globally unique identifier, which is linked to the memory address of the underlying `View` object (see Section 2.2 for more details on how UI hierarchies are represented). Toller-enhanced testing tools can subsequently use these identifiers to precisely specify the UI element that should be executed. This feature is especially helpful when (1) an app's UI is constantly changing and UI events from the tools cannot be easily executed on the desired UI element, and (2) a tool wants to execute UI events on not-easily accessible UI elements such as list items that are visible only after scrolling. In general, we find that Toller with UI Hierarchy Capturing and UI Event Execution has better testing effectiveness than Toller with just UI Hierarchy Capturing (see

Table 2: UI Hierarchy Capturing (Capt) efficiency comparison

	$\textsc{Ch}_O$	$\textsc{Ch}_E$	$\textsc{Wt}_O$	$\textsc{Wt}_E$	$\textsc{St}_O$	$\textsc{St}_E$
Time per capt	846	22	360	82	8175	245
# of capt	31182	47599	47666	83603	1348	47525

Notes: Ch, Wt, and St denote Chimp, WCTester, and Stoat, respectively. For each tool T, T_O refers to its original version, while T_E refers to our Toller-enhanced version. Time is shown in milliseconds.

Table 3: UI Event Execution (Exec) efficiency comparison

	$\textsc{Ch}_O$	$\textsc{Ch}_E$	$\textsc{Wt}_O$	$\textsc{Wt}_E$	$\textsc{St}_O$	$\textsc{St}_E$
Time per exec	762	454	455	372	8395	391
# of exec	22045	57714	51137	65340	2118	44466

Notes: Ch, Wt, and St denote Chimp, WCTester, and Stoat, respectively. For each tool T, T_O refers to its original version, while T_E refers to our Toller-enhanced version. Time is shown in milliseconds.

Section 5.6 for more details). Therefore, in all of our experiments except Section 5.6, we define Toller-enhanced as Toller with both UI Hierarchy Capturing and UI Event Execution.

5 EVALUATION

To understand the impact that Toller's infrastructure enhancements can have on Android testing tools, we investigate five main research questions:
RQ1: How does each of Toller's infrastructure enhancements contribute to Android testing tools' efficiency?
RQ2: How does enhancing Android testing tools with Toller improve achieved code coverage?
RQ3: How does enhancing Android testing tools with Toller improve achieved crash triggering ability?
RQ4: How much do covered code entities and triggered crashes overlap for each tool that is and is not enhanced with Toller, respectively?
RQ5: How does each of Toller's infrastructure enhancements contribute to Android testing tools' effectiveness?

We address RQ1 to understand how Toller's various infrastructure enhancements affect the run time performance benefits of Toller. We address RQ2 and RQ3 to understand how Toller affects Android testing tools on two metrics commonly used to evaluate such tools and to understand whether the effectiveness rankings of these tools change when they are and are not enhanced with Toller, respectively. We address RQ4 to understand the extent that a tool enhanced with Toller covers the same code entities and triggers the same crashes as the tool not enhanced with Toller. Finally, we address RQ5 to understand how Toller's various infrastructure enhancements affect two metrics commonly used to evaluate Android testing tools.

5.1 Evaluation Setup

To answer our RQs, we use the same experiment environment and set of apps as our motivating study (Section 3.1). We collect the method coverage as code coverage achieved by each run using the MiniTrace [17] tool, which modifies DalvikVM/ART and does not require app instrumentation. We consider only crashes originated from app bytecode and collect code locations in stack traces as crash signatures. We obtain stack traces by monitoring and filtering Android Logcat [16] messages. As mentioned in Section 4, we use Toller to remove apps' `UncaughtExceptionHandlers` to ensure that stack traces are being reported to Logcat.

In addition to the three Android UI testing tools used in our motivating study (Section 3.2), we also use *Ape* [18], another state-of-the-art tool, for our experiments. We use the four tools in the following settings.

- Both WCTester and Stoat run on computers and use UIAutomator to capture UI hierarchies. To enhance the two tools with Toller while keeping implementation changes minimal, we translate Toller's captured UI hierarchies to the UIAutomator's format to make them directly readable by these two tools. For UI Event Execution, WCTester injects low-level UI events directly using ADB shell commands, while Stoat sends UI element queries to UIAutomator to generate and inject the corresponding low-level UI events. We replace both tools' original implementation of UI Event Execution with Toller.
- Like WCTester and Stoat, Chimp also runs on computers and can use UIAutomator to capture UI hierarchies. To enhance Chimp with Toller, we choose to fully incorporate Toller into Chimp to avoid unnecessary translations of UI hierarchies. For UI Event Execution, Chimp injects low-level UI events directly using ADB shell commands while its Toller-enhanced version uses Toller for UI Event Execution.
- As the most recently proposed state-of-the-art tool in our experiments, Ape already provides support for fast UI Hierarchy Capturing in its implementation by using hidden Android accessibility service APIs. These implementation details are not explicitly discussed in the tool's paper [18]. To show the significance of infrastructure support on Ape, we modify Ape to build its slow version, which leverages UIAutomator services in the same way as the other tools. We then compare the slow version's testing effectiveness with the original version of Ape. Note that unlike the other Toller-enhanced tools, the original Ape contains only the UI Hierarchy Capturing enhancement and not the UI Event Execution enhancement because UI Event Execution requires Toller's UI Hierarchy Capturing (Section 4).

In total, we have nine tool versions in our experiments: Chimp (with and without Toller), WCTester (with and without Toller), Stoat (with and without Toller), Ape (original and slow version), and Monkey. To compensate for potential randomness in our experiments introduced by tool or app logic, we run each tool on each app three times, with each run being one hour. Overall, we spend 27 hours per app (9 tool versions * 3 runs for each version) and a total of 405 hours (27 * 15 apps) for all apps.

5.2 RQ1: Efficiency of Enhancements

To understand how Toller's UI Hierarchy Capturing and UI Event Execution infrastructure enhancements affect the run time performance benefits of Toller, we integrate Toller with the three testing tools from Section 3.1 and re-run the experiments in the same settings. The time usage statistics of UI Hierarchy Capturing

Table 4: Average method coverage for all tool versions

App Name	Ape$_S$	Ape$_O$	Δ	Mk	Ch$_O$	Ch$_E$	Δ	Wt$_O$	Wt$_E$	Δ	St$_O$	St$_E$	Δ
Abs	8273	8424	1.8%	6213	6656	6874	3.3%	7527	7622	1.3%	3538	5282	49.3%
Duolingo	14180	14598	2.9%	8948	13500	13447	-0.4%	11659	13482	15.6%	6297	14006	122.4%
Filters For Selfie	2369	5489	131.7%	4077	2290	2262	-1.2%	2608	2170	-16.8%	2345	2227	-5.0%
GoodRx	15524	14272	-8.1%	13149	14033	15499	10.4%	13829	15904	15.0%	8716	12243	40.5%
Google Translate	8918	9169	2.8%	7554	7376	8477	14.9%	8793	8960	1.9%	3653	6855	87.7%
Marvel Comics	5306	5873	10.7%	4538	4672	4781	2.3%	4378	4460	1.9%	3459	4908	41.9%
Merriam-Webster	8229	8287	0.7%	5141	7657	8486	10.8%	6661	8241	23.7%	7238	7856	8.5%
Mirror	1120	1124	0.4%	426	1253	657	-47.6%	1105	851	-23.0%	887	948	6.9%
My Baby Piano	1096	3419	212.0%	165	1583	1652	4.4%	1373	1553	13.1%	700	219	-68.7%
Sketch	7695	8124	5.6%	6871	8081	8532	5.6%	7666	7782	1.5%	5755	7766	34.9%
trivago	19491	20079	3.0%	19678	18721	19317	3.2%	19373	19164	-1.1%	4325	19424	349.1%
WEBTOON	20415	24493	20.0%	19775	12590	21457	70.4%	14329	23338	62.9%	5695	13672	140.1%
Word	12057	12387	2.7%	11911	10875	12713	16.9%	12834	12612	-1.7%	8445	11482	36.0%
Youtube	28026	24888	-11.2%	17945	17086	18123	6.1%	17923	18930	5.6%	11162	18434	65.1%
Zedge	32937	43080	30.8%	28139	35532	38786	9.2%	34530	36665	6.2%	19490	30664	57.3%
Average	**12376**	**13580**	**9.7%**	**10302**	**10794**	**12071**	**11.8%**	**10973**	**12116**	**10.4%**	**6114**	**10399**	**70.1%**

Notes: Mk, Ch, Wt, and St denote Monkey, Chimp, WCTester, and Stoat, respectively. For each tool T, T$_O$ refers to its original version, while T$_E$ refers to our Toller-enhanced version. Ape$_S$ refers to the slow version of Ape. Each integer cell shows the average number of covered distinct methods across three runs by the corresponding tool version on the corresponding app. For each tool T, $\Delta = (\mathrm{T}_E - \mathrm{T}_O)/\mathrm{T}_O \times 100\%$. Average $\Delta = (\overline{\mathrm{T}}_E - \overline{\mathrm{T}}_O)/\overline{\mathrm{T}}_O \times 100\% = (\Sigma\mathrm{T}_E - \Sigma\mathrm{T}_O)/\Sigma\mathrm{T}_O \times 100\%$.

and UI Event Execution are shown in Tables 2 and 3, respectively. Note that the numbers for each tool are aggregated from running on all 15 apps once. As shown in Tables 2 and 3, Toller is capable of reducing overheads for both primitive interfaces on all three tools. Specifically, the average time usages for UI Hierarchy Capturing are reduced by about 97%, 77%, and 97% on Chimp, WCTester, and Stoat, respectively. For UI Event Execution, the average time usages are reduced by 40%, 18%, and 95% on Chimp, WCTester, and Stoat, respectively. We find that UI Event Execution has less substantial overhead reductions than UI Hierarchy Capturing because as described in Section 4, Toller falls back on using low-level UI event injections on event handlers that have not been covered by a low-level UI event injection.

UI Hierarchy Capturing and UI Event Execution can take a substantially different amount of time for different tools because some of the tools use different approaches to invoke UIAutomator (e.g., directly invoking the `uiautomator` command in the ABD shell as used by Chimp, or using a service wrapper [22] for ease of programming in the case of WCTester and Stoat). Another observation is that the original implementation of Stoat takes much more time than the other tools to perform both types of operations. We find that this result is related to how Stoat uses the UIAutomator service wrapper: Stoat's implementation essentially sets up and establishes new connections to the on-device service agent before *each* capture or action. On the contrary, WCTester sets up this connection only once and persists the connection, eliminating much overhead.

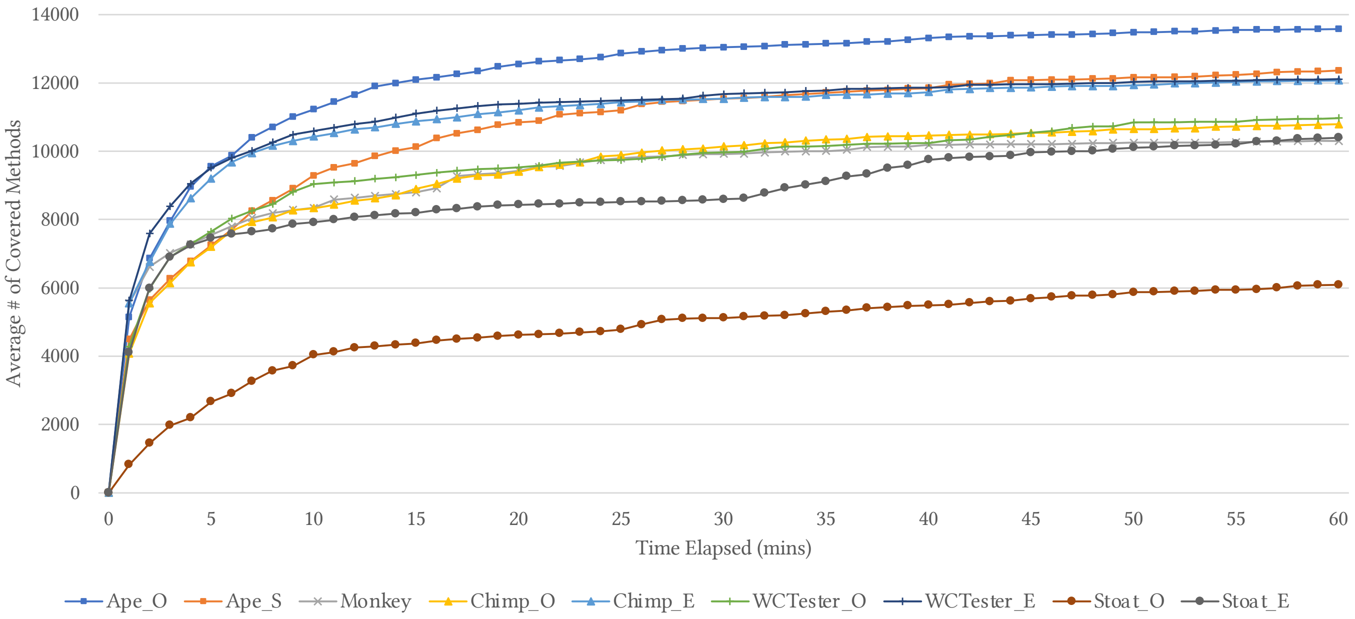
5.3 RQ2: Code Coverage Benefits

Table 4 shows the average coverage statistics of each pair of tools and apps from our experiments. Figure 4 shows the changes of average code coverage across all apps for each tool along with run time. Note that methods that can be covered after app launch but before testing starts are excluded. As shown in Table 4 and Figure 4, using Toller's infrastructure enhancements helps improve the testing effectiveness of various Android UI testing tools. Specifically, enhancing Chimp, WCTester, and Stoat with Toller yields 11.8%, 10.4%, and 70.1% average method coverage improvements, respectively. For Ape, the tool's own fast UI Hierarchy Capturing implementation brings 9.7% average method coverage improvement. It should also be noted that the aforementioned percentages are calculated based on the average number of covered methods across different apps, where apps with a larger codebase can have a bigger impact on the results.

One key finding is that the differences of code coverage brought by infrastructure enhancements can be substantial enough to change the relative competitiveness among tools. For example, Table 4 shows that for Stoat, compared with Monkey, the Toller-enhanced version achieves higher average code coverage, while the original version achieves lower average code coverage. In Ape's case, the slow version has much smaller advantages over other tools: its average code coverage is only about 2% relatively higher than the Toller-enhanced WCTester. We also find that the original Ape achieves higher code coverage than other tools on 10 apps, while the slow Ape does that on only 4 apps. For comparison, the Toller-enhanced Chimp and WCTester top the ranks on 4 and 3 apps, respectively, when we omit the original Ape from consideration. Figure 4 additionally shows that the slow Ape (denoted as Ape$_S$) constantly has comparable average code coverage as the Toller-enhanced WCTester (denoted as WCTester$_E$), where Ape$_S$ starts to beat WCTester$_E$ after 40 minutes of testing. Interestingly, according to Figure 4, the Toller-enhanced WCTester even has higher code coverage than all other tools in the first several minutes of testing. Specifically, when we observe the area under the curve (AUC) for every minute, we find that if developers are given at most 9 minutes to use Android UI testing tools, then the Toller-enhanced WCTester gives the highest AUC instead of the original Ape.

5.3.1 *Analysis of Negative Code Coverage Improvements.*

To better understand our results for this RQ, we manually study some of our



Notes: Each data point shows how many methods have been covered on average across three runs on all apps by the respective tool, after the corresponding amount of time has elapsed in each run. The ending number of covered methods for each tool is the same as that in the "Average" row in Table 4.

Figure 4: Average method coverage by elapsed time during testing

Table 5: Cumulative numbers of distinct crashes for all tool versions

App Name	Ape$_S$	%$_S$	Ape$_O$	%$_O$	ΣApe	MK	Ch$_O$	%$_O$	Ch$_E$	%$_E$	ΣCh	Wt$_O$	%$_O$	Wt$_E$	%$_E$	ΣWt	St$_O$	%$_O$	St$_E$	%$_E$	ΣSt
Abs	1	33%	2	67%	3	3	1	100%			1	3	75%	1	25%	4	8	67%	12	100%	12
Duolingo	1	50%	1	50%	2			-		-				1	100%	1	6	55%	9	82%	11
Filters For Selfie		-		-		1		-		-			-		-		3	75%	4	100%	4
GoodRx			1	100%	1				5	100%	5	1	13%	7	88%	8	6	67%	5	56%	9
Google Translate			1	100%	1				1	100%	1		-		-		8	73%	5	45%	11
Marvel Comics	1	100%			1				1	100%	1			1	100%	1	9	82%	9	82%	11
Merriam-Webster		-		-				-		-			-		-		4	44%	9	100%	9
Mirror	3	60%	5	100%	5	5	3	60%	5	100%	5	5	83%	4	67%	6	5	63%	7	88%	8
My Baby Piano		-		-				-		-			-		-			-		-	
Sketch		-		-				-		-			-		-		4	80%	4	80%	5
trivago	1	33%	2	67%	3	3	1	100%			1			1	100%	1	8	53%	11	73%	15
WEBTOON			1	100%	1	1		-		-		1	100%			1	8	57%	14	100%	14
Word			1	100%	1	2			4	100%	4	1	33%	2	67%	3	6	55%	11	100%	11
Youtube		-		-					1	100%	1	2	100%			2	13	59%	16	73%	22
Zedge	1	100%			1				1	100%	1			3	100%	3	4	40%	9	90%	10
Total	**8**	**42%**	**14**	**74%**	**19**	**15**	**5**	**25%**	**18**	**90%**	**20**	**13**	**43%**	**20**	**67%**	**30**	**92**	**61%**	**125**	**82%**	**152**

Notes: Ch, Wt, and St denote Chimp, WCTester, and Stoat, respectively. For each tool T, T$_O$ refers to its original version, while T$_E$ refers to our Toller-enhanced version. Each integer cell under T$_O$ or T$_E$ shows the cumulative number of distinct crashes across three runs by the corresponding tool version on the corresponding app. A blank cell indicates no crash. Each integer cell under ΣT indicates the union number of covered methods by the two tool versions on the corresponding app. $\%_O = \mathrm{T}_O/\Sigma\mathrm{T} \times 100\%$, similar for $\%_E$. In the 'Total' row, each integer indicates the sum value of all numbers in the respective column, while each percentage is calculated from sum values using the same methodology as aforementioned.

results to understand why some tools have negative code coverage improvements on some apps after Toller's infrastructure enhancements. Specifically, we manually look into all of the cases where the coverage decrement is over 1% given that smaller changes (<1%) are likely caused by random noise. We look at tool logs and differences in method coverage to speculate root causes.

We are able to identify only one major cause for the reduced effectiveness: *Unsupported UI element types.* The current implementation of Toller does not support obtaining the inner contents of certain types of UI elements, such as `WebViews` that maintain their own non-standard, internal UI-related data structures. These `WebViews` are a major cause for why apps such as "Filters For Selfie" and "Mirror" have negative code coverage improvements for WCTester. Both of these two apps have Google's AdMob SDK embedded and the SDK relies on `WebViews` to display ads. Without knowing the UI hierarchy inside, it is difficult for tools to produce meaningful UI events to fully exercise this ads-related logic. Future work should explore how Toller can better handle certain types of UI elements (e.g., falling back to UIAutomator for `WebViews`).

Table 6: Distribution of exception types for all tool versions

Exception Type	Ape_S	Ape_O	Mk	Ch_O	Ch_E	Wt_O	Wt_E	St_O	St_E	Total
`ActivityNotFoundException`	2	5	4	2	4	4	3		3	**11**
`ExceptionInInitializerError`								2	2	**3**
`IllegalArgumentException`					1				1	**2**
`IllegalStateException`	2	3	4		2		4		2	**17**
`NoClassDefFoundError`								2	2	**2**
`NullPointerException`	1	4	2	1	5		9	5	11	**25**
`OutOfMemoryError`			1			2				**3**
`RuntimeException`	2	2	4	2	3	6	2	81	102	**129**
Other	1				3	1	2	2	2	**11**
Total	**8**	**14**	**15**	**5**	**18**	**13**	**20**	**92**	**125**	**203**

Notes: Each cell shows the number of distinct crashes of the specific type triggered by the corresponding tool on all apps. A blank cell indicates no crash. If an exception type appears only once across all tools and apps, it is counted in "Other" instead of being shown in a separate row. Thus, the numbers of distinct crashes in "Other" also indicate the numbers of exception types. The "Total" column shows the numbers of *distinct* crashes for each exception type across all tools.

Table 7: Cumulative method coverage for all tool versions

App Name	Ape_S	$\%_S$	Ape_O	$\%_O$	ΣApe	Ch_O	$\%_O$	Ch_E	$\%_E$	ΣCh	Wt_O	$\%_O$	Wt_E	$\%_E$	ΣWt	St_O	$\%_O$	St_E	$\%_E$	ΣSt
Abs	8872	95%	9318	100%	9348	9077	97%	7479	80%	9369	9107	89%	8205	80%	10206	6367	77%	7380	89%	8290
Duolingo	14789	96%	15018	97%	15471	14282	97%	14235	96%	14770	12086	81%	14136	94%	14989	12397	80%	14937	96%	15558
Filters For Selfie	2380	42%	5684	99%	5730	2343	97%	2286	95%	2407	2759	68%	2177	54%	4067	2479	73%	3365	99%	3406
GoodRx	16451	96%	15504	90%	17191	14973	89%	16214	97%	16762	14383	85%	16615	98%	16911	12257	81%	14173	93%	15185
Google Translate	9709	93%	10292	99%	10428	8545	79%	10537	97%	10829	9732	95%	9689	95%	10192	6519	72%	7665	85%	9000
Marvel Comics	5614	81%	6688	96%	6946	5016	98%	4940	96%	5140	4736	94%	4614	92%	5031	4667	83%	5344	95%	5647
Merriam-Webster	8776	97%	8614	95%	9046	8293	90%	8858	96%	9223	6859	75%	8710	95%	9192	8353	86%	9314	95%	9766
Mirror	1196	93%	1253	97%	1290	1514	99%	796	52%	1534	1309	97%	959	71%	1354	1058	79%	1213	91%	1335
My Baby Piano	1572	31%	5059	100%	5065	1590	87%	1810	100%	1818	2682	66%	1555	38%	4083	1654	100%	253	15%	1662
Sketch	8737	92%	8919	94%	9519	8687	93%	8955	96%	9311	8120	91%	8264	93%	8913	6874	71%	9351	97%	9614
trivago	19999	97%	20437	100%	20524	19857	98%	19980	98%	20342	20010	98%	19847	97%	20399	6072	29%	20926	100%	20981
WEBTOON	23982	85%	27149	97%	28088	18048	63%	27933	98%	28643	19754	68%	27185	93%	29238	9957	36%	25123	91%	27457
Word	13550	93%	13034	90%	14514	11711	78%	14645	97%	15095	13946	94%	13706	93%	14763	11055	80%	13493	98%	13768
Youtube	31010	87%	28268	79%	35681	21122	86%	21427	87%	24572	20101	77%	23709	91%	26069	20866	71%	22451	76%	29427
Zedge	39932	77%	50763	98%	51562	52210	93%	41536	74%	56212	39612	95%	38550	92%	41765	28960	68%	36159	85%	42503
Average	**13771**	**86%**	**15067**	**94%**	**16027**	**13151**	**87%**	**13442**	**89%**	**15068**	**12346**	**85%**	**13195**	**91%**	**14478**	**9302**	**65%**	**12743**	**89%**	**14240**

Notes: Ch, Wt, and St denote Chimp, WCTester, and Stoat, respectively. For each tool T, T_O refers to its original version, while T_E refers to our Toller-enhanced version. Each integer cell under T_O or T_E shows the cumulative number of covered distinct methods across three runs by the corresponding tool version on the corresponding app. Each integer cell under ΣT indicates the union number of covered methods by two tool versions on the corresponding app. $\%_O = \text{T}_O/\Sigma\text{T} \times 100\%$, similar for $\%_E$. In the 'Average' row, each integer indicates the average value of all numbers in the respective column, while each percentage is calculated from average values using the same methodology as aforementioned.

5.4 RQ3: Crash Triggering Benefits

Table 5 shows the cumulative number of distinct crashes (from three runs) for all tool versions evaluated on each app. As shown in the table, the Toller-enhanced versions are capable of substantially improving the total number of distinct crashes, from 5, 13, and 92 to 18, 20, and 125 for Chimp, WCTester, and Stoat, respectively. In Ape's case, the total crash count rises from 8 to 14 by using Ape's improved infrastructure support. Overall, we find that there are 43 pairs of tools and apps with at least one crash (non-empty cells under ΣT). Of the 43 pairs, 30 and 10 pairs have more and fewer (respectively) crashes triggered by enhanced tool versions than original/slow tool versions. The remaining 3 pairs have the same number of crashes for both tool versions. Of the 30 pairs where the enhanced tool versions have more crashes, 21 pairs' cumulative crashes are all from the enhanced tool versions (highlighted cells under T_E and Ape_O). On the other hand, of the 10 pairs where the enhanced tool versions have fewer crashes, only 6 pairs' cumulative crashes are all from the original/slow versions. In general, when the original/slow versions trigger more crashes, the differences are generally small: only one crash for 7 of the 10 pairs. Randomness in the tools' and apps' logic is likely responsible for why original/slow versions can trigger more distinct crashes than Toller-enhanced versions. Overall, our results find that infrastructure enhancements help testing tools with not only covering more code but also triggering more distinct crashes.

5.4.1 *Exception Types.*

We additionally study the distribution of exception types. As shown in Table 6, the Toller-enhanced versions trigger not only more instances of crashes, but also more types of exceptions (the number of non-empty cells): from 3, 4, and 6 types to 8, 6, and 9 types on Chimp, WCTester, and Stoat, respectively. In Ape's case, the slow version triggers 5 types of exceptions, while the original version triggers only 4 types. One possible explanation for this finding is the randomness in the tools' and apps' logic. Nevertheless, our results still find that infrastructure enhancements help most testing tools trigger more distinct types of crashes.

5.5 RQ4: Overlap of Code Coverage and Crashes

In RQ4, we investigate whether the code coverage achieved and crashes triggered by tools with infrastructure enhancements are subsumed by what the original/slow versions of the tools achieve

Table 8: Average method coverage for enhancements

	None	HC Only	Δ	HC + EE	Δ
Chimp	10794	12072	11.8%	12071	11.8%
WCTester	10973	12112	10.4%	12116	10.4%
Stoat	6114	10121	65.5%	10399	70.1%

Table 9: # of distinct crashes for enhancements

	None	HC Only	Δ	HC + EE	Δ
Chimp	5	8	1.6x	18	3.6x
WCTester	13	15	1.2x	20	1.5x
Stoat	92	103	1.1x	125	1.4x

Notes: "None" denotes that no infrastructure enhancement is applied. "HC Only" denotes that only UI Hierarchy Capturing enhancement is used, while "HC + EE" denotes that both UI Hierarchy Capturing and UI Event Execution enhancements are used.

and trigger. To answer this RQ, we measure the overlaps between code coverage achieved and distinct crashes triggered by both versions of each tool.

Table 7 shows the cumulative code coverage for all tool versions. Specifically, for each tool on each app, this table shows how many methods are covered by either version in any run, as well as how many methods can be covered by only one of the versions. As shown in the table, the Toller-enhanced versions achieve, on average, 89%, 91%, and 89% coverage of all methods that can be covered by either version of Chimp, WCTester, and Stoat, respectively. In Ape's case, the original version achieves 94% coverage of all methods that can be covered by either the slow or original version. Our results suggest that tool versions with infrastructure enhancements can generally replace the original versions as the versions with enhancements provide the most of the coverage achievable by either version.

We use the same methodology to show the overlaps of crashes triggered by the two versions of each tool. As shown in Table 5, the Toller-enhanced versions also cover most of the crashes triggered by either version shown by the $\%_E$ columns in the table. In fact, 90%, 67%, and 82% of the cumulative distinct crashes detected by Chimp, WCTester, and Stoat, respectively, are triggered by the Toller versions. For Ape, the original version covers 74% of all crashes. Our results again suggest that tool versions with infrastructure enhancements can generally replace the original versions as the enhanced versions provide the most of the detected crashes.

5.6 RQ5: Effectiveness of Enhancements

To understand how Toller's two enhancements have contributed to testing tools' code coverage and crash triggering ability, we additionally conduct experiments by enabling only UI Hierarchy Capturing and comparing its results with the setting where both enhancements are used (results in Sections 5.3 and 5.4). We do not evaluate only UI Event Execution, as Toller's UI Event Execution implementation depends on UI Hierarchy Capturing and does not work on its own as discussed in Section 4. Tables 8 and 9 show the average method coverage and the number of distinct crashes, respectively, for all testing tools using different enhancement options under the same experimental settings. More detailed experiment data is available on our website [1].

As shown in Tables 8 and 9, enhancing UI Hierarchy Capturing already improves both achieved code coverage and triggered crashes, while enhancing UI Event Execution leads to even better testing effectiveness, particularly for the number of distinct crashes triggered. One interesting finding is that for Chimp and WCTester, the UI Event Execution enhancement does not improve the overall code coverage. Beyond the fact that all tools are likely to cover less new code as time increases, we identify multiple additional causes for why UI Event Execution may not increase code coverage:

- As discussed in Section 4, UI Event Execution skips the logic of dispatching low-level UI events on UI elements, likely resulting in the loss of coverage. We mitigate this limitation by falling back to low-level UI event injection when we find an event handler that has not been exercised. However, the strategy could still miss edge cases (e.g., when different UI elements share the same parameterized event handler class).
- Faster UI Event Execution and faster UI Hierarchy Capturing can result in higher CPU usages and overload the emulators, likely causing apps to stop responding.
- Tools might not be accustomed to both fast UI Event Execution and fast UI Hierarchy Capturing. Being unaccustomed to both may cause the tools to be too fast when an app loads content asynchronously, and the time overhead incurred by slower UI Event Execution or UI Hierarchy Capturing actually helps the tools properly wait for the content to load. Such cases are known to cause UI flaky tests [35].

Future work should explore how to carefully design solutions to address the aforementioned causes. For example, future work can intelligently decide on the waiting time at each step to mitigate the effects of device overloading or asynchronous loading with minimal unnecessary waiting costs.

6 THREATS TO VALIDITY

The internal threats to the validity of our work are that Toller's implementation and the scripts used to generate the tables and figures could have faults that might have affected our results. Furthermore, our setup of the Android testing tools used in our experiments could have been incorrect and affected our results. To mitigate these internal threats to validity, we design our experiments to output extensive logs along with the metrics used in our experiments. We then manually analyze a sample of the logs from our experiments to ensure that the presented results match what we observe from the logs.

The main external threat to the validity of our work is the representativeness of the apps and the Android UI testing tools selected for our experiments. To mitigate this threat to validity, we select the top apps from 13 different categories of apps on the Google Play Store. Therefore, the selected apps vary greatly in their functionality and APK size (from 3.3MB to 93MB). The Android testing tools used for our experiments are from a previous study [38] of Android UI testing tools. The study finds that Monkey is the best Android UI testing tool among six tools. Among these tools, we find that two are runnable on our infrastructure and do not require app instrumentation. To demonstrate the improvements that Toller can have on Android UI testing tools, particularly on ones that use UIAutomator, we select the two tools from the previous study and implement a version of Monkey, known as Chimp, that uses UIAutomator for our experiments.

Another threat to the validity of our work is the randomness from the Android UI testing tools, apps, and emulators. Namely, across different runs of the same tool, app, and emulator, the obtained metrics could change. To mitigate this threat to validity, we run each pair of tools and apps three times, where each run is performed on a newly-created emulator with the same software and hardware configurations throughout all of the experiments. The conclusions that we make from our results are then from the aggregation of the three runs for each pair of tools and apps.

7 DISCUSSION

Modifying Android OS. For our experiments, we modify AOSP Android 6.0 on both emulators and real devices. Our modified emulator image is publicly available [1] as a portable testing environment for others to immediately begin using.

While modifying the Android framework eliminates the risks of app instrumentation, it is true that modifying the Android framework can also be undesirable. For instance, we might fail to modify a customized Android OS. Additionally, the modification usually requires root access to the testing device, not being always feasible. To support developers who may be interested in infrastructure enhancements without modifying the Android framework, we also design Toller so that it can be bundled with the target app's code through source code integration or binary instrumentation. Because Toller relies on only Android framework classes, it is only necessary to inject a startup method call into existing app code to make Toller work.

On the other hand, we argue that different testing needs should be satisfied with different ways of support. Specifically, the Android-framework-based solution is suitable for external testing on certain devices, such as app examinations conducted by app marketplaces. The code-bundling-based solution may be more suitable for in-house testing conducted by app developers, such as compatibility testing that involves various devices.

Future Work. Our work demonstrates how Android UI testing can be improved with infrastructure enhancements, but there are many other future directions to improve mobile app testing and debugging. Based on the current implementation of Toller, we can additionally support ultra-low overhead UI event monitoring, which is essential for capture and replay tools [27]. Toller can also be extended to support performing UI manipulation (e.g., disabling a button) at runtime, allowing testers to conveniently control the regions of interest for any tool. Toller even has the potential of assisting with in-situ static analysis by providing the bytecode and argument values of any UI event handler at runtime, allowing testing tools to predict the consequences of any UI event without destroying the current app state. The aforementioned future directions are just some of many examples of how infrastructure enhancements can improve testing and debugging of mobile apps.

8 RELATED WORK

Automated UI testing for Android. Our work tackles the efficiency issue of infrastructure support for automated Android UI testing tools. A number of automated UI testing tools have been published over the years. One of the earliest efforts is Monkey [15], a tool from Google, originally intended for stress testing of app UIs. While receiving almost no feedback from the target app, Monkey still manages to outperform many research tools with its high event generation and execution efficiency. Subsequent efforts have led to tools based on randomness/evolution [29, 31, 39], UI modeling [6, 9, 18, 21, 28, 37], and systematic exploration [2, 4, 30].

UI capture and replay for Android. Our work also directly benefits many Android UI capture and replay tools that often require UI Hierarchy Capturing and UI Event Execution. Existing Android UI capture and replay tools can be categorized into coordinate-based [3, 11, 24, 32, 33] and UI-element-based [5, 8, 10, 19, 20, 33, 34] tools. The latter tool category is likely to have very low efficiency [27], e.g., needing seconds to capture a single action, similar to the Android UI testing tools used in our work.

Infrastructure support for Android testing. There has been work trying to improve infrastructure support for the purpose of efficient testing. Hu et al. [23] proposed work that aims to quickly find potential sequences of error-triggering UI inputs through direct invocations of UI event handlers, achieved by instrumenting the target apps. Song et al. [36] also proposed a similar idea. Toller's UI Event Execution support achieves similar goals. It should be noted that Toller aims to provide infrastructure support for any tool in need of either UI Hierarchy Capturing or UI Event Execution. Additionally, Toller does not require app instrumentation, which often breaks functionalities, especially on industrial apps.

9 CONCLUSION

Much work has been proposed by researchers to improve Android UI testing tools with sophisticated algorithmic designs. Recent studies have shown that these tools barely outperform (w.r.t. code coverage and crash triggering ability) Monkey, a simple tool that generates and injects purely randomized UI events. To understand the inefficiencies of Android testing tools, we have conducted a motivating study to determine the sources and extents of the inefficiencies for these tools. Our motivating study has found that capturing information about the contents on the screen (*UI Hierarchy Capturing*) and executing UI events (*UI Event Execution*, such as clicks) use on average 70% of the testing run time. Based on our findings, we have proposed Toller, a tool to provide efficient infrastructure support for UI Hierarchy Capturing and UI Event Execution to Android UI testing tools. Our experiments show that Toller can substantially (1) reduce the run time used by the infrastructure that the testing tools depend on and (2) improve the code coverage and crash triggering ability of these tools when they are given a reasonable amount of run time. We make the source code of Toller and the scripts used to set up Toller publicly available [1]. We hope that our results can raise the community's awareness of the significance of infrastructure support beyond the community's existing heavy focus on algorithms.

ACKNOWLEDGMENTS

We thank Lingming Zhang and Tianyin Xu for discussions about this work. This work was partially supported by 3M Foundation Fellowship, Google CMD-IT LEAP fellowship, and NSF grants CNS-1564274, CCF-1763788, and CCF-1816615.

REFERENCES

[1] Toller artifacts, 2021. https://github.com/TOLLER-Android/main.
[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. FSE, 2012.
[3] appetizer-toolkit, 2017. https://github.com/appetizerio/appetizer-toolkit.
[4] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. OOPSLA, 2013.
[5] Bot-bot, 2021. http://imaginea.github.io/bot-bot/index.html.
[6] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. OOPSLA, 2013.
[7] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? ASE, 2015.
[8] Culebra, 2021. https://github.com/dtmilano/AndroidViewClient/wiki/culebra.
[9] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury. Time-travel testing of Android apps. ICSE, 2020.
[10] Espresso test recorder, 2021. https://developer.android.com/studio/test/espresso-test-recorder.html.
[11] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. ICSE, 2013.
[12] Google. Android 64K method limit, 2021. https://developer.android.com/studio/build/multidex.
[13] Google. Android apps on Play Store, 2021. https://play.google.com/store/apps.
[14] Google. Android debug bridge (ADB), 2021. https://developer.android.com/studio/command-line/adb.
[15] Google. Android Monkey, 2021. https://developer.android.com/studio/test/monkey.
[16] Google. Logcat command-line tool, 2021. https://developer.android.com/studio/command-line/logcat.
[17] T. Gu. Minitrace, 2021. http://gutianxiao.com/ape.
[18] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su. Practical GUI testing of Android applications via model abstraction and refinement. ICSE, 2019.
[19] J. Guo, S. Li, J.-G. Lou, Z. Yang, and T. Liu. Sara: Self-replay augmented record and replay for Android in industrial cases. ISSTA, 2019.
[20] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: Cross-platform user-interaction record and replay for the fragmented Android ecosystem. 2015.
[21] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. MobiSys, 2014.
[22] X. He. Python wrapper of Android uiautomator test tool, 2021. https://github.com/xiaocong/uiautomator.
[23] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. EuroSys, 2014.
[24] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for Android. OOPSLA, 2015.
[25] IDC and Gartner. Share of Android OS of global smartphone shipments from 1st quarter 2011 to 2nd quarter 2018. https://www.statista.com/statistics/236027/global-smartphone-os-market-share-of-android.
[26] JesusFreke. smali/baksmali, 2021. https://github.com/JesusFreke/smali.
[27] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie. Record and replay for Android: Are we there yet in industrial cases? ESEC/FSE, 2017.
[28] Y. Li, Z. Yang, Y. Guo, and X. Chen. DroidBot: A lightweight UI-guided test input generator for Android. ICSE-C, 2017.
[29] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. ESEC/FSE, 2013.
[30] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. FSE, 2014.
[31] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for Android applications. ISSTA, 2016.
[32] Z. Qin, Y. Tang, E. Novak, and Q. Li. MobiPlay: A remote execution based record-and-replay tool for mobile applications. ICSE, 2016.
[33] Ranorex, 2021. http://www.ranorex.com/mobile-automation-testing.html.
[34] Robotium recorder, 2021. https://robotium.com/products/robotium-recorder.
[35] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang. An empirical analysis of UI-based flaky tests. ICSE, 2021.
[36] W. Song, X. Qian, and J. Huang. EHBDroid: Beyond GUI testing for Android applications. ASE, 2017.
[37] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based GUI testing of Android apps. ESEC/FSE, 2017.
[38] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie. An empirical study of Android test generation tools in industrial cases. ASE, 2018.
[39] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android apps with intent-filter tag. MoMM, 2013.
[40] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie. Automated test input generation for Android: Are we really there yet in an industrial case? FSE, 2016.
[41] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie. Automated test input generation for Android: Towards getting there in an industrial case. ICSE-SEIP, 2017.