VET: Identifying and Avoiding UI Exploration Tarpits

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

Wei Yang
University of Texas at Dallas, USA
wei.yang@utdallas.edu

Tianyin Xu
University of Illinois at Urbana-Champaign, USA
tyxu@illinois.edu

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

ABSTRACT

Despite over a decade of research, it is still challenging for mobile UI testing tools to achieve satisfactory effectiveness, especially on industrial apps with rich features and large code bases. Our experiences suggest that existing mobile UI testing tools are prone to *exploration tarpits*, where the tools get stuck with a small fraction of app functionalities for an extensive amount of time. For example, a tool logs out an app at early stages without being able to log back in, and since then the tool gets stuck with exploring the app's pre-login functionalities (i.e., exploration tarpits) instead of its main functionalities. While tool vendors/users can manually hardcode rules for the tools to avoid specific exploration tarpits, these rules can hardly generalize, being fragile in face of diverted testing environments, fast app iterations, and the demand of batch testing product lines. To identify and resolve exploration tarpits, we propose VET, a general approach including a supporting system for the given specific Android UI testing tool on the given specific app under test (AUT). VET runs the tool on the AUT for some time and records UI traces, based on which VET identifies exploration tarpits by recognizing their patterns in the UI traces. VET then pinpoints the actions (e.g., clicking logout) or the screens that lead to or exhibit exploration tarpits. In subsequent test runs, VET guides the testing tool to prevent or recover from exploration tarpits. From our evaluation with state-of-the-art Android UI testing tools on popular industrial apps, VET identifies exploration tarpits that cost up to 98.6% testing time budget. These exploration tarpits reveal not only limitations in UI exploration strategies but also defects in tool implementations. VET automatically addresses the identified exploration tarpits, enabling each evaluated tool to achieve higher code coverage and improve crash-triggering capabilities.

CCS CONCEPTS

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

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

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468554

KEYWORDS

UI testing, trace analysis, mobile testing, mobile app, Android

ACM Reference Format:
Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. VET: Identifying and Avoiding UI Exploration Tarpits. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468554

1 INTRODUCTION

With the prosperity of mobile apps [19], especially their roles in people's daily life during pandemic (e.g., food ordering, grocery delivery, and social networking), quality assurance of mobile apps becomes crucially important. User Interfaces (UIs), as the primary interface of user-app interactions, are natural entry points for app testing. While manual UI testing is still often used in practice, automated UI testing is becoming popular [21, 29, 33, 36]. Automated UI testing mimics how human users interact with apps through the UIs and detects reliability and usability issues. Automated UI testing complements manual testing with greater timing flexibility and better code coverage, requiring little human intervention.

However, existing mobile UI testing tools are found to be ineffective in exploring app functionalities, despite their sophisticated strategies for UI exploration. While recent proposals [1, 2, 7, 18, 22, 24–26, 35, 43, 44] have reported promising results, measurement studies on comprehensive app benchmarks [8, 39] have drawn different conclusions. For example, a recent study [39] shows that state-of-the-art mobile UI testing tools yield low code coverage (about 30% in method coverage) after hours of testing on popular industrial apps. Note that industrial apps typically have richer functionalities and larger code bases, compared with open-source apps. The findings suggest a significant effectiveness gap that needs to be filled for automated mobile UI testing.

According to our experience, the ineffectiveness of existing mobile UI testing tools often stems from their proneness to *exploration tarpits*[1], where tools get stuck with a small fraction of app functionalities for an extensive amount of time. We show a real-world example in §2, where a state-of-the-art Android UI testing tool named Ape [17] decides to log itself out one minute after testing an app starts, without being able to log back in, and since then gets stuck with exploring the app's pre-login functionalities (i.e., exploration tarpits) instead of its main functionalities. It is possible that tool vendors/users manually hardcode rules for the tools to avoid specific exploration tarpits, such as instructing Ape to avoid

[1]The name of exploration tarpits is inspired by the Mythical Man-Month book [4].

tapping the "logout" button or writing a script to support automatic login. However, these rules can hardly generalize, being fragile in face of diverted testing environments (e.g., unreliable network to process login requests), fast app iterations, and the demand of batch testing product lines. Our findings in §5.2 show various cases as such where exploration tarpits can be caused by unexpected flaws in a tool's exploration strategies and/or implementation defects.

To automatically identify and resolve exploration tarpits, in this paper, we propose a general approach and its supporting system named Vet for the given specific Android UI testing tool on the given specific app under test (AUT). Vet works in three stages. (1) Vet runs the tool on the AUT for some time and records the interactions between the tool and AUT, in the form of UI *traces*. A UI trace consists of app UIs interleaving with the actions taken by the tool. (2) Vet then analyzes the collected traces to identify trace subsequences (termed *regions*) that manifest exploration tarpits. (3) Vet guides the tool in subsequent test runs to prevent or recover from an exploration tarpit by monitoring the testing progress and taking actions based on findings from the identified regions.

Vet includes two specialized algorithms targeting two corresponding patterns of exploration tarpits: *Exploration Space Partition* and *Excessive Local Exploration* (see §2 and §4). Exploration Space Partition, corresponding to Figure 1a, indicates that the fraction of app functionalities explored by the tool is disconnected from most of the app functionalities after some specific action (e.g., tapping "OK" in Screen C). Such situations can be prevented by disabling the aforementioned action. Excessive Local Exploration indicates that the tool enters a hard-to-escape fraction of the app UIs and needs a significant amount of time to reach other functionalities, as demonstrated in Figure 1b. This issue can be addressed by either preventing the tool from entering (e.g., disabling "START" in Screen E in Figure 1b) or assisting the tool to escape (e.g., restart the app upon observation of Screen F). To design the two algorithms, we first construct fitness value formulas that quantify how well a region on the given trace matches a targeted pattern. We then apply fitness value optimization on the entire trace to determine the region that best fits our targeted patterns.

We evaluate Vet using three state-of-the-art/practice Android UI testing tools (Monkey [12], Ape [17], and WCTester [45, 53]) with 16 widely used industrial apps. We collect 144 traces by running each tool on each app three times for one hour each (*original* runs). Vet reports at least one exploration tarpit region in each (tool, app) pair, with 131 regions in total, each spanning about 27 minutes on average. The longest regions span over 59 minutes, about 98.6% of the one-hour testing time budget. After inspecting the 131 reported regions, we confirm the root causes of 96 regions, including both limitations of UI exploration strategies (e.g., early logouts) and defects in tool implementation (e.g., hanging), as shown in §5.2. We then perform six other one-hour runs for each (tool, app) pair: (1) three *guided* runs using Vet to automatically avoid all the exploration tarpit regions identified in the original runs during testing on three runs, and (2) three *comparison* runs not using Vet.

Based on the preceding evaluation setup, we compare the code coverage (of the given app) achieved by applying each tool with and without the assistance of Vet given the same time budget. Specifically, we compare the combined code coverage and the numbers of distinct crashes for (1) original runs and guided runs, and (2) original runs and comparison runs. The evaluation results show that on average a tool assisted by Vet achieves up to a 15.3% relative code coverage increment and triggers up to 2.1x distinct crashes than the tool without the assistance of Vet.

In summary, this paper makes the following main contributions:

- A new perspective of improving the given automated UI testing tool by automatically identifying and addressing exploration tarpits for the given target AUT;
- Algorithms for effective identification of two manifestation patterns of exploration tarpits;
- A practical system [37] that can be automatically applied to enhance any Android UI testing tool such as Monkey [12], Ape [17], and WCTester [45, 53], on any AUT;
- Comprehensive evaluation of Vet, demonstrating that Vet reveals various issues related to tools or app usability, and that Vet automatically resolves those issues, helping the tools achieve up to a 15.3% relative code coverage increment and 2.1x distinct crashes on 16 popular industrial apps.

2 MOTIVATING EXAMPLES

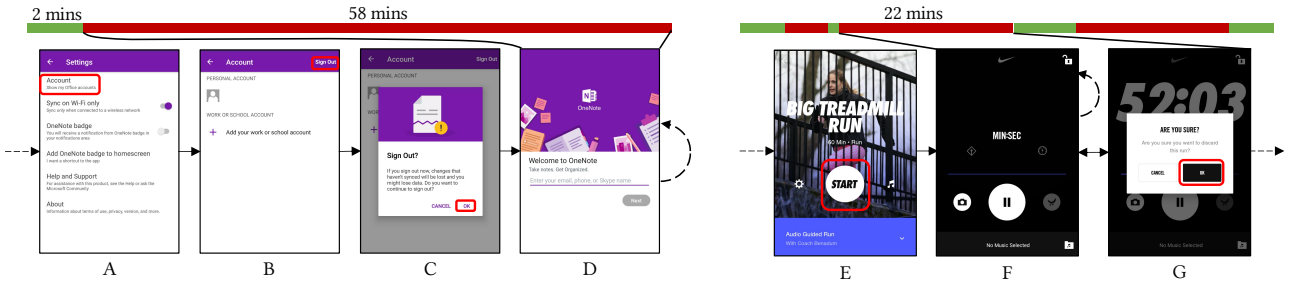
We present two concrete examples from our experiments covering Exploration Space Partition and Excessive Local Exploration (see §4). These examples provide contexts for further discussion and help illustrate the motivations that drive the design of Vet.

2.1 Exploration Space Partition

We run Ape [17], a state-of-the-art Android UI testing tool to test a popular app, Microsoft *OneNote*. The result is illustrated in Figure 1a. We manually set up the account to log in to the app's main functionalities, and then start Ape. We run Ape without interruptions for one hour and check the test results afterward.

In the one-hour testing period, Ape explores only 12% (9 out of 76) activities. To understand the low testing effectiveness, we investigate the UI trace captured during testing and find the root cause to be exploration tarpits:

(1) Ape performs exploration around OneNote's main functionalities for about two minutes, covering 7 (out of 9) of all the activities covered in the entire one-hour test run. We omit this phase in Figure 1a.
(2) About two minutes after testing starts, Ape arrives at the "Settings" screen (Screen A) and decides to click "Account" (the red-boxed UI element) for further exploration.
(3) Ape arrives at the "Account" screen (Screen B) and clicks the "Sign Out" button. The click pops up a window (Screen C) asking for confirmation of getting logged out.
(4) Ape clicks "CANCEL" first, and then goes back to the "Account" screen. However, Ape clicks the "Sign Out" button again, knowing that there is one action not triggered yet in the confirmation dialog. Subsequently, Ape clicks the "OK" button (Screen C) and logs itself out.
(5) The logout leads to the entry screen (Screen D). From this point, Ape has access to only a small number of functionalities (e.g., logging in). Ape cannot log in due to the difficulty of auto-generating the username/password of the test account. In the remaining 58 minutes, Ape explores two new activities in total.



(a) Exploration Space Partition (b) Excessive Local Exploration

Note: Colored bars on the top represent the progress of two 1-hour tests, where green bars refer to normal exploration and red bars refer to exploration tarpits. Dashed straight arrows indicate visiting the screen from some other screen, and solid arrows show transitions between two screens after clicking the red-boxed UI elements. The dashed curve arrow on Screen D depicts that Ape cycles around D until the end of testing. The dashed curve arrow on Screen F shows that Monkey stays on F within the 22-minute exploration tarpit window.

Figure 1: Motivating examples of exploration tarpits described in §2.

This example represents Exploration Space Partition described in §1. The essential problem is that Ape does not understand UI semantics—it does not know that the majority of OneNote's functionalities will be unreachable by clicking the "OK" button at the time of action.

2.2 Excessive Local Exploration

Figure 1b presents another example in which we run Monkey [12], a widely adopted tool, to test another popular industry-quality app, *Nike Run Club*. In this example, Monkey spends about 22 minutes trying to saturate one of the app's functionalities. After investigating into the collected UI trace, we find the following behavior when Monkey interacts with the app:

(1) Monkey explores other functionalities normally before entering Screen E that allows the tool to enter the functionality where the tool later gets trapped. We name the functionality *the trapping functionality*. Monkey clicks the "START" button and enters the trapping functionality (Screen F).
(2) Monkey keeps clicking around in the trapping functionality. To escape from the trapping functionality, Monkey first needs to press the Back button, and a confirmation dialog (Screen G) will pop up. Monkey then has to click the "OK" button to finish escaping. However, due to being widget-oblivion, Monkey clicks only randomly on the screen, resulting in constant failures to click "OK" when the confirmation dialog is shown. Furthermore, the dialog disappears when Monkey clicks outside of its boundary, and Monkey needs to press the Back button again to make the confirmation show up one more time. It takes 22 minutes for Monkey to find and execute an effective escaping UI event sequence and finally leave the trapping functionality.
(3) The aforementioned behaviors are repeatedly observed in the trace (with different amounts of time used for escaping).

This example represents Excessive Local Exploration behavior described in §1. The essential problem is that Monkey is both widget- and state-oblivion, i.e., the tool is unable to locate actionable UI elements efficiently or sense whether it has been trapped and react accordingly (e.g., by restarting the target app).

2.3 Implications

To prevent such undesirable exploration behaviors, a conceptually simple idea is to de-prioritize exploring the entries to aforementioned trapping states (i.e., the "OK" button in Screen C, and "START" button in Screen F). One potential solution is to develop natural language processing (NLP) or image processing based approaches that can infer the semantics of UI elements [23, 30, 41, 42]. While solutions based on understanding UI semantics are revolutionary, they are challenging due to fundamental difficulties rooting in NLP and image processing.

In this paper, we explore a more practical and evolutionary solution based on understanding exploration tarpits by mining UI traces. We show that it is feasible to identify the existence and location of such behavior through pattern analysis on interaction history. Given the location of exploration tarpits, we can further identify which UI actions might have led to such behavior. Taking the example of Figure 1, Ape starts to visit a very different set of screens (e.g., the welcome screens in Screen D in Figure 1a) after clicking "OK", and the number of explored screens dramatically decreases. Therefore, we can look at the screen history and find the time point where the symptom starts to appear. The UI action located at the aforementioned time point is then likely the cause of the symptom. Our VET system uses a specialized algorithm (§4.2) to effectively locate the starting time point of exploration tarpits similar to the aforementioned instance.

3 BACKGROUND

This section presents background knowledge about UI hierarchy to help readers understand our algorithm design and implementations in the scope of Android UI testing.

A *UI hierarchy* structurally represents the contents of app UI shown at a time. Each UI hierarchy consists of *UI properties* (e.g., location, size) for individual *UI elements* (e.g., buttons, textboxes) and hierarchical relations among UI elements. On Android, each activity internally maintains the data structure for its current UI hierarchy. Typically, UI elements are represented by `View` [13] subclass instances, and hierarchical relations are represented by child `Views` of `ViewGroup` [14] subclass instances.

A key component of UI testing is to identify the current app functionality. The functionality is identified by *equivalence check for UI hierarchies*, because UI hierarchies are usually used as indicators of apps' functionality scenarios. Thus, checking the equivalence between the current and past UI hierarchies allows tools to identify whether a new functionality is being exercised. If the current functionality has been covered, the tool can additionally leverage the knowledge associated with the functionality to decide on the next actions. There are different ways to check UI hierarchy equivalence:

- **Strict comparison.** A simple way to check the equivalence of two UI hierarchies is to compare their UI element trees and see whether they have identical structure and UI properties at each node. In practice, such simple equivalence checking is too strict. For example, on an app accepting text inputs, a tool checking exact equivalence can count a new functionality every time one character is typed.
- **Checking similarity.** A workaround to the aforementioned issue of strict comparison is to check similarities of two UI hierarchies against a threshold. However, ambiguity can become the new issue, given that the similarity relation is not transitive: suppose that A is similar to both B and C, it is still possible that B is not similar to C. Then if both B and C are in the history (regarded as different functionalities), and A comes as a new UI hierarchy, the tool is unable to decide on which functionality to use the associated knowledge from. To fix the ambiguity issue, we can perform *screen clustering*, essentially putting mutually similar screens into individual groups and regarding each group as representing one single functionality. Then the downside is that screen clustering can be a computationally expensive operation, especially for traces with many screens.
- **Comparing abstractions.** A more advanced solution is to check the equivalence at an abstraction level, employed by many model-based UI testing tools [3, 7, 16, 17, 35]. In the previous example, one can leave out all user-controlled textual UI properties from the hierarchy and the equivalence check can tell that the tool is staying on the same screen regardless of what has been entered. While abstracting UI hierarchies is conceptually effective, it is challenging to design effective UI abstraction functions. The difficulty lies in identifying UI properties or structural information to distinguish different app functionalities, especially when screens have variants with relatively subtle differences.

  Ape [17] includes adaptive abstractions to address the challenge of automatically finding proper UI abstraction functions in different scenarios. Ape dynamically adjusts its abstraction strategy (e.g., which UI property values should be preserved) during testing based on feedback from strategy execution (e.g., whether invoking actions on UI hierarchies with the same abstraction yields the same results). Unfortunately, the adaptive abstraction idea assumes the availability of sufficiently diversified execution history for feedback, and such history is not always available when analyzing given traces as in our situation.

Given the pros and cons of the aforementioned ways, we empirically adopt a hybrid approach for UI hierarchy equivalence check. First, we always abstract UI hierarchies: (1) we consider only visible UI elements (i.e., `View.getVisibility() == VISIBLE`, and the element's bounding box intersects with its parent's screen region), (2) we keep the activity ID and the original hierarchical relations among UI elements, and (3) we retain only UI element types and IDs from UI properties. Second, we check the similarities of abstract UI hierarchies and cluster them into groups only when the analysis is sensitive to the absolute number of distinct screens. More details on achieving clustering efficiently are elaborated in §4.3.

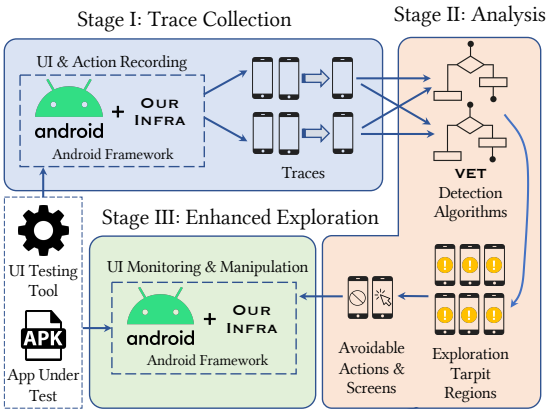


Figure 2: Overview of VET.

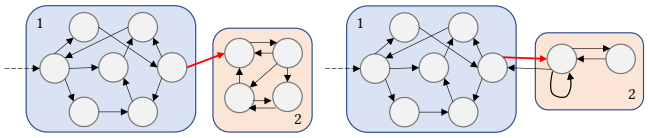
4 THE VET APPROACH

4.1 Overview

We propose VET, a general approach and its supporting system that automatically identifies and addresses exploration tarpits for *any* given Android UI testing tool on *any* given AUT. Our implementation of VET is publicly available at [37].

As illustrated in Figure 2, for a given tool and AUT, VET works in three stages. First, VET runs the target tool on the AUT for a certain amount of time and records the interactions between the tool and AUT. With help from our Android framework extension TOLLER [38], VET collects *trace*(s) that consist of AUT UIs interleaving with the tool's actions. Then, VET analyzes each individual trace with specialized algorithms to identify trace subsequences (termed *regions*) that manifest the tool's exploration tarpits. Optionally, one can rank the identified regions based on their time lengths, where longer regions receive higher ranks, to prioritize regions that are likely to exhibit exploration tarpits with higher impacts (see §5.2). Finally, VET learns from the identified regions and guides the tool in subsequent runs to avoid exploration tarpits, by monitoring the testing progress and taking actions based on findings from the identified regions. With the support from TOLLER, VET is currently capable of (1) preventing specified actions by disabling the corresponding UI elements at runtime and (2) assisting the AUT to escape from the specified screens by restarting the AUT. The identified regions additionally support manual investigations of testing efficacy by providing localization help.

We equip VET with two specialized algorithms targeting two patterns of exploration tarpits: *Exploration Space Partition* and *Excessive Local Exploration*. Characteristics of the two algorithms' targeted patterns are illustrated in Figure 3 and discussed as follows:



Note that each subgraph corresponds to an example trace, where each circle represents a distinct screen in the trace (e.g., each subfigure in Figure 1), each arrow indicates that action(s) is observed between two screens in the trace, and each curved rectangle depicts a UI subspace. Red arrows denote destructive actions (e.g., clicking "OK" in Screen C of Figure 1a, "START" in Screen E of Figure 1b) while dashed arrows show where traces begin.

Figure 3: Two patterns of exploration tarpits.

- *Exploration Space Partition.* As shown in Figure 3a, the UI testing tool traverses through a UI subspace (Subspace 2) for a long time after the execution of some action (the red arrow), and the tool is *unable* to return to the previously visited UI subspace (Subspace 1). Furthermore, the tool visits much fewer distinct screens after the action. The presence of the symptom suggests that the tool has triggered a destructive action (effectively the partition boundary of the entire trace and beginning of the exploration tarpit) that prevents the tool from further exploring the app's major functionalities. The first motivating example from §2 corresponds to this symptom, where clicking the "OK" button is the destructive action that gets Ape trapped in multiple screens related to logging in (Subspace 2) and prevents Ape from further accessing OneNote's main functionalities (Subspace 1).
- *Excessive Local Exploration.* As shown in Figure 3b, the UI testing tool is trapped in a small UI subspace (Subspace 2) for an extended amount of time after the execution of the corresponding destructive action (the red arrow). However, the tool is *capable* of returning to the previously visited UI subspace (Subspace 1) despite the difficulties. It is also likely that the tool will get trapped again within Subspace 2 after returning to Subspace 1. Consequently, the tool spends an excessive amount of time repetitively testing limited functionalities in this hard-to-escape subspace. The second motivating example from §2 corresponds to this symptom, where clicking the "START" button gets Monkey trapped in Screens F and G (Subspace 2). Clicking "OK" helps Monkey go back to Screen E (within Subspace 1) and other functionalities, but it does not take a long time before the tool gets trapped again within Subspace 2.

As can be seen, Exploration Space Partition targets higher-level irreversible transition of UI exploration space, while Excessive Local Exploration focuses on lower-level difficulties of exercising a specific functionality. Note that it is possible for the regions reported by the two algorithms on the same trace to overlap. For example, Excessive Local Exploration might also capture exploration tarpits within Exploration Space Partition's trapped UI subspace (corresponding to Subspace 2 in Figure 3). Such overlaps do not prevent us from finding meaningful targeted exploration tarpits: different exploration tarpits revealed by regions identified by both algorithms suggest the existence of different exploration difficulties.

Table 1: Notations and descriptions used in the algorithms

Notation	*Description*
S_i	Screen #i in the trace represented by the UI hierarchy.
t_i	The timestamp of screen S_i being observed.
$S_{l,r}$	A region of screens starting at Screen #l and ending at Screen #r (with both ends included).
$\{S_{l,r}\}$	The set of distinct screens from $S_{l,r}$ by de-duplicating their UI hierarchies.
$\lvert S^s_{l,r}\rvert$	The number of occurrences of s in $S_{l,r}$.
$t_{\min}$	A predefined threshold that decides the minimum time length of any $S_{l,r}$ (i.e., $t_r - t_l \geqslant t_{\min}$) that may be included in algorithm outputs.

In the remaining of this section, we describe the two algorithms for capturing Exploration Space Partition and Excessive Local Exploration in §4.2 and §4.3, respectively. We show that pattern capturing can be expressed as optimization problems. Table 1 describes the notations used to describe VET's algorithms.

4.2 Capturing Exploration Space Partition

According to our introductions of Exploration Space Partition, we need to find a destructive action exerted on screen S_n as the partition boundary such that the aforementioned characteristics from §4.1 can be best reflected. For instance, considering our first motivating example in §2, we hope to pick up the screen shown in Figure 1c as S_n. We optimize the following formula to find the most desirable S_n from a trace with N screens:

$$\underset{1\leqslant n<E_p}{\arg\min} \quad [\sum_{s\in\{S_{1,n}\}} \frac{|S^s_{n+1,N}|}{N-n}] + 2\cdot\sigma(\frac{|\{S_{n+1,N}\}|}{|\{S_{E_p+1,N}\}|} - 1) - 1$$

In the formula, E_p is a pre-calculated limit indicating the upper bound of n during optimization, and σ denotes the Sigmoid function. Note that $N-n$ can be pulled out of the sum subformula. The intuition of the formula design is as follows:

(1) As part of the characteristics, the tool should ideally be able to visit few to no screens that have appeared no later than S_n after the tool passes S_n. Correspondingly, in our motivating example, screens shown before Figure 1c (depicting the app's main functionalities) are dramatically different from the screens afterward (logging in, ToS, etc.). In the formula, the nominator of the first term (intended to be minimized) quantifies the proportion of screens seen before S_n within $S_{n+1,N}$.
(2) As the denominator of the first term, $N-n$ essentially calculates how many (non-distinct) screens the tool visits after S_n. There are two purposes of this design. First, we hope to normalize the first term in the formula (so that two terms can weigh the same). Given that $\sum_{s\in\{S_{1,n}\}} |S^s_{n+1,N}| = \sum_{s\in\{S_{1,n}\}\cap\{S_{n+1,N}\}} |S^s_{n+1,N}| \leqslant \sum_{s\in\{S_{n+1,N}\}} |S^s_{n+1,N}| = N-n$, the first term is guaranteed to fall within $[0,1]$. Second, we want to push S_n backward (note that smaller n makes the first term smaller) because we assume that the design makes S_n closer to the exploration tarpit's root cause, which should appear earlier than other causes.

(3) As another part of the characteristics, the tool stays within a certain UI subspace for a long time; thus, the tool will go through screens within the subspace very often. If the tool generally uniformly visits most or all distinct screens within the subspace, by observing a small period of exploration (corresponding to $S_{E_p+1,N}$ in the formula) we should have a fairly precise estimation (i.e., $\{S_{E_p+1,N}\}$) of the subspace boundary, which is characterized by $\{S_{n+1,N}\}$. The second term in the formula corresponds to this intuition, where the closer $\{S_{E_p+1,N}\}$ is to $\{S_{n+1,N}\}$ (note that $\{S_{E_p+1,N}\} \subseteq \{S_{n+1,N}\}$), the more favorable it becomes during optimization.
(4) By setting an upper bound E_p on n and regularizing the ratio with a Sigmoid function and applying appropriate linear transformations, we can guarantee that the second term in the formula always ranges from 0 to 1, being the same as the first term. In the end, two terms in the formula contribute equally to optimization choices.

To determine E_p on each trace, because our optimization scope does not include any interval shorter than $[E_p, N]$, we choose a value such that $t_N - t_{E_p}$ is closest to $t_{\min}$.

After obtaining a potentially suitable S_n through optimizing the aforementioned formula, we additionally check whether $|\{S_{1,n}\}| > |\{S_{n+1,N}\}|$ is satisfied, essentially enforcing the property that the exploration space should be smaller after the partition. Finally, the reported region is $S_{n+1,N}$.

4.3 Capturing Excessive Local Exploration

Based on the characteristics of Excessive Local Exploration from §4.1, we should track the presence of a region showing that the tool is trapped within a small UI subspace for an extended amount of time. For our second motivating example in §2, one valid choice is the 22-minute region starting from the button click in Screen E of Figure 1b. We accordingly optimize the following formula to find the boundaries S_l and S_r of the most suitable region on a trace:

$$\arg\min_{1 \leqslant l \leqslant r \leqslant N} \frac{|\textsc{Merge}(\{S_{l,r}\})|}{r - l + 1}$$

In the formula, Merge denotes the operation of merging similar screens and returning the groups of merged screens. As the optimization formula suggests, we hope to find a suitable region such that it covers few distinct screen groups despite that the tool tries to explore diligently (by injecting numerous actions quantified by $r - l + 1$). Then if $t_r - t_l \geqslant t_{\min}$, we regard that the exploration tarpit region $S_{l,r}$ can be reported. Accordingly in our motivating example, the choice of S_l is Screen F of Figure 1b and S_r is the last instance of Screen G of Figure 1b in the 22-minute region. S_{l-1} corresponds to Screen E of Figure 1b, and the destructive action is reported.

Note that there can be more than one region exhibiting Excessive Local Exploration behavior within a single trace, given the possibility for the tool to escape the UI subspaces where Excessive Local Exploration behavior is observed. In order to find all potential regions, we iterate the aforementioned optimization process on remaining region(s) each time after one region is chosen, until no more region can be divided.

Design of Merge. As mentioned in §3, involving screen merging is especially useful for handling Excessive Local Exploration, given that the aforementioned optimization formula is very sensitive to the absolute numbers of distinct screens. Being part of the challenge, an efficient (and mostly effective) screen merging algorithm requires careful design. Given a set of distinct abstract screens (represented by UI hierarchies) to merge, a relatively straightforward (and precise) approach is to first calculate the tree editing distance [46] for each pair of abstract UI hierarchies for similarity check, and then use combinatorial optimization [40] to decide the optimal grouping strategy (e.g., by converting to an Integer Linear Programming [31] problem), such that all screens within the same group are mutually similar and the total number of groups is minimal. Unfortunately, such an algorithm requires exponential time in regards to the number of distinct abstract screens to merge. The design will likely fall short on traces collected using industrial-quality apps, from which we can easily capture hundreds to thousands of distinct abstract screens.

Aiming to make the algorithm practically efficient, we relax the definition of similarity and the goal of optimization from the aforementioned merging algorithm based on insights from our observations. Specifically, we find that in many cases, similar screens can be seen as screen variants derived from base screens by inserting a small number of leaf nodes or subtrees into the abstract UI hierarchy. Based on this assumption with some tolerance for inaccuracy, we can (1) design a more efficient tree similarity checker (Algorithm 2), which considers only node insertion distances and has $\Theta(|h_1| \cdot |h_2|)$ time complexity (compared with $O(|h_1| \cdot |h_2| \cdot \textsc{Height}(h_1) \cdot \textsc{Height}(h_2))$ for full tree edit distance), and (2) replace the inefficient combinatorial optimization with a highly efficient greedy algorithm (Algorithm 1), which tries to find all the base screens with $O(|H|^2 \cdot \max_{h \in H} |h|^2)$ time complexity. In practice, with multiple other optimizations not affecting the level of time complexity, the algorithm needs only several seconds on average to process a trace. Even for a very long trace with 2,000 distinct abstract screens and tens of thousands of concrete screens, the algorithm runs for only several minutes.

Input: A set of abstract UI hierarchies H
Output: A mapping $R : H \mapsto R$, where $R \subseteq H$
Sort $h \in H$ by $|h|$ in ascending order
$R \leftarrow \{\}$
foreach $h \in H$ **do**
  $R[h] \leftarrow$ nil
end
foreach $h \in H$ **do**
  if $R[h] = nil$ **then**
    $R[h] \leftarrow h$
    foreach $h' \in H$ **do**
      if $R[h'] =$ nil $\wedge$ SimCheck(h, h') **then**
        $R[h'] \leftarrow h$
      end
    end
  end
end
return R

Algorithm 1: Merge: Merging similar screens into groups (each group is represented by its root screen in R)

Input: Abstract UI hierarchies h_1, h_2
Output: Whether h_1, h_2 are similar enough
Const: Max allowed distance $d_{\max}$, empirically set to 3
$seq_1 \leftarrow [\,]$
foreach $node \in \text{DepthFirstTraverse}(h_1)$ **do**
  $seq_1 \leftarrow seq_1 :: (\text{Props}(node), \text{Depth}(node))$
  // Props obtains the node's UI properties that are preserved during abstraction. See more details in §3.
end
$seq_2 \leftarrow [\,]$
foreach $node \in \text{DepthFirstTraverse}(h_2)$ **do**
  $seq_2 \leftarrow seq_2 :: (\text{Props}(node), \text{Depth}(node))$
end
$lcs \leftarrow \text{LongestCommonSequence}(seq_1, seq_2)$
if $|lcs| < \min(|h_1|, |h_2|)$ **then**
  return `false`
else
  return $\max(|h_1|, |h_2|) - |lcs| \leqslant d_{\max}$
end

Algorithm 2: SimCheck: UI hierarchy similarity checker

Table 2: Overview of industrial apps used for evaluation

App Name	Version	Category	#Inst	Login
AccuWeather	5.3.5-free	Weather	50m+	✗
AutoScout24	9.3.14	Auto & Vehicles	10m+	✗
Duolingo	3.75.1	Education	100m+	✗
Flipboard	4.1.1	News & Magazines	500m+	✓
Merriam-Webster	4.1.2	Books & Reference	10m+	✗
Nike Run Club	2.14.1	Health & Fitness	10m+	✓
OneNote	16.0.9126	Business	100m+	✓
Quizlet	3.15.2	Education	10m+	✓
Spotify	8.4.48	Music & Audio	100m+	✓
TripAdvisor	25.6.1	Food & Drink	100m+	✓
trivago	4.9.4	Travel & Local	10m+	✗
Wattpad	6.82.0	Books & Reference	100m+	✓
WEBTOON	2.4.3	Comics	10m+	✗
Wish	4.16.5	Shopping	100m+	✓
YouTube	15.35.42	Video Player & Editor	1b+	✗
Zedge	7.2.2	Personalization	100m+	✗

Notes: '#Inst' denotes the approximate number of downloads. 'Login' indicates whether the app requires logging in to access most features.

5 EVALUATION

Our evaluation answers the following research questions:

- **RQ1:** How effectively can VET help reveal Android UI testing tool issues with the identified exploration tarpit regions?
- **RQ2:** What is the extent of effectiveness improvement of Android UI testing tools through automatic enhancement by VET?
- **RQ3:** How likely do VET algorithms miss tool issues in their identified exploration tarpit regions?

5.1 Evaluation Setup

Android UI Testing Tools and Android Apps. We use three state-of-the-art/practice Android UI testing tools: Monkey [12], Ape [17], and WCTester [45, 53]. We use 16 popular industry Android apps from the Google Play Store, as shown in Table 2. These 16 apps are from a previous study [39], which picks the most popular apps from each of the categories on Google Play and compares multiple testing tools applied on these apps. The apps that we choose need to work properly on our testing infrastructure: (1) they need to provide x86/x64 variants of native libraries (if they have any), (2) they do not constantly crash on our emulators, and (3) TOLLER is able to obtain UI hierarchies on most of the functionalities. We additionally skip apps that (1) have relatively limited sets of functionalities, or (2) require logging in for access to most features and we are unable to obtain a consistently usable test account (e.g., some apps have disabled our test accounts after some experiments).

Trace Collection. We run each tool on every app for three times to alleviate the potential impacts of non-determinism in testing. Each run takes one hour without interruption, and we restart the tool if it exits before using up the allocated run time. TOLLER records one UI trace for each test run. While VET runs separately on each UI trace, results are grouped for each (tool, app) pair. In total, we collect 144 one-hour UI traces from 48 (tool, app) pairs.

Testing Platform. All experiments are conducted on the official Android x64 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. 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. We manually write auto-login scripts for apps with "Login" ticked in Table 2, and each of these scripts is executed only once before the corresponding app starts to be tested in each run. To alleviate the flakiness of these auto-login scripts, we manually check the collected traces afterward and rerun the experiments with failed login attempts.

Overall Statistics. VET reports 131 regions to exhibit exploration tarpits, averaging 2.7 on each (tool, app) pair. Based on VET's reports, the average amount of time involved in exploration tarpits is about 27 minutes per region, with the maximum being 59 minutes and minimum being slightly more than 10 minutes (given that we empirically set $t_{\min} = 10$ minutes for all the experiments).

5.2 RQ1. Detected Tool Issues

5.2.1 Methodology. We evaluate the effectiveness of VET algorithms in capturing exploration tarpits that reveal issues of testing tools upon AUTs. Specifically, we first group exploration tarpit regions by the (tool, app) pairs that these regions are observed on. Then, we rank the regions within each (tool, app) pair by their time lengths as mentioned in §4.1. Finally, we manually investigate each of 131 regions from all (tool, app) pairs. We report any issue for each of these regions with manual judgment. Note that we count only the issue that we consider most specific to the exploration tarpit revealed by each region: if both issues A and B contribute to the exploration tarpit on some region, and A also contributes to other regions on the same trace, we count only B in the statistics.

5.2.2 Results. We are able to determine tool issues on 96 of 131 manually investigated regions. Table 3 shows the distribution of issue types w.r.t the tool and region ranking. Note that we find each (tool, app) pair to have up to three regions reported by VET; thus, rank-1/2/3 regions cover all 131 regions (with 48/43/40 regions each). The tool issues can be traced to two root causes: *apps under test require extra knowledge for effective testing*, and *tool defects prevent themselves from progressing*. We discuss specific issues w.r.t. these two root causes:

Table 3: Distribution of confirmed issue types

Issue	Rank-1				Rank-2				Rank-3				Total
	Ape	Mk	Wt	Sum	Ape	Mk	Wt	Sum	Ape	Mk	Wt	Sum	
LOUT	5	2	1	8	5	2	1	8	4	1	1	6	**22**
UI	0	4	0	4	0	3	0	3	2	4	0	6	**13**
NTWK	0	5	0	5	0	3	0	3	0	3	0	3	**11**
LOOP	0	0	7	7	0	0	8	8	0	0	5	5	**20**
ESC	0	4	0	4	0	2	0	2	0	2	0	2	**8**
ABS	0	0	2	2	0	0	2	2	0	0	1	1	**5**
MISC	5	1	2	8	5	1	0	6	3	0	0	3	**17**
Total	**10**	**16**	**12**	**38**	**10**	**11**	**11**	**32**	**9**	**10**	**7**	**26**	**96**

Notes: 'Mk' and 'Wt' refer to Monkey and WCTester, respectively. Issue type details are discussed in §5.2.

- **App logout or equivalent** (abbreviated as "LOUT" in Table 3) accounts for exploration tarpits on 23% (22 out of 96) of investigated regions with identified issues. Some apps essentially require login states for the majority of their functionalities to be accessible. However, the tools used in our experiments have no knowledge about the consequences of clicking the "logout" buttons in different apps before the tools actually try clicking these buttons. Unfortunately, after the tools try out such actions (driven by their exploration strategies), the apps' login states (both on-device and on-server) have been destroyed. The tools then have to spend all remaining time on a limited number of functionalities, leading to exploration tarpits. This case reveals a common weakness of existing UI testing tools—they have a limited understanding of action semantics. As can be seen from Table 3, Ape is more likely to be affected by this type of issues.
- **Unresponsive UIs** ("UI") are found in 14% (13 out of 96) of regions. We find that some apps stop responding to UI actions after the advertisement banner is clicked, even though the apps' UI threads are not blocked. The issue is likely caused by the UI design defects in the Google AdMob SDK. The AUTs should be restarted as soon as possible to resume access to their functionalities. According to Table 3, Monkey is most vulnerable to such issues. One interesting finding is that Ape is actually also vulnerable to unresponsive UIs although the tool's implementation is capable of identifying such situations. However, while Ape proceeds to restart the app most of the times when Ape finds the app unresponsive, Ape fails to do so occasionally.
- **Network disconnections** ("NTWK") are found in 11% (11 out of 96) of regions. We find that turning networking off is undesirable for some apps, especially when the disconnection lasts for a long time. Consequently, these apps may show only messages prompting users to check their networks, leaving nothing for exploration. Tools such as Monkey can control network connections through Android system UI (e.g., by clicking the "Airplane Mode" icon). While the capability helps test app logic in edge conditions in general, it might hurt the tool's effectiveness on apps heavily relying on network access. All regions with the aforementioned issue come from Monkey's traces.
- **Restart/action loops** ("LOOP") are found in 21% (20 out of 96) of regions. The tool essentially keeps restarting or performing the same actions on the target app after some point. One potential cause for such issues is that the tool thinks that *all* UI elements in the target app's main screen have been explored. The tool might need to revise its exploration strategy for discovering more explorable functionalities.
- **Obscure escapes** ("ESC") are found in 8% (8 out of 96) of regions. Defects in a tool's design or implementation can make it difficult for the tool to escape from certain app functionalities, and consequently, the tool loses opportunities to explore other functionalities. For instance, Monkey finds it challenging to escape a screen where the only exit is a tiny button on the screen (see the second motivating example in §2), due to the tool's lack of understanding of UI hierarchies.
- **UI abstraction defects** ("ABS") are found in 5% (5 out of 96) of regions. Defects in a tool's UI abstraction strategies can trick the tool into incorrectly understanding the testing progress. Seen from collected traces, WCTester considers all texts as part of abstract UI hierarchies. While the strategy works well in a wide range of testing scenarios, it keeps the tool repetitively triggering actions on UI elements with changing texts (such as counting down), given that the tool incorrectly thinks that new functionalities are being covered.
- **Miscellaneous tool implementation defects** ("MISC") are in 18% (17 out of 96) of regions. In our case, we find potential implementation defects in three tools: (1) Ape and Monkey fail to handle unresponsive apps ("Injection failed" or being unable to obtain UI hierarchies), and (2) WCTester is found to explore only a certain fraction of app functionalities after some point.

Another finding is that the ratio of confirmed issues decreases when the rank goes lower (38/48 = 79% for rank-1, 32/43 = 74% for rank-2, and 26/40 = 65% for rank-3), suggesting the usefulness of prioritizing regions based on their lengths.

5.3 RQ2. Improvement of Testing Performance

This section shows that the identified exploration tarpit regions by Vet can be used to automatically address tool issues.

5.3.1 Automatic fix application. The essential idea is to prevent some tool issues from happening again or getting rid of tool issues quickly by controlling the interactions between tools and apps. Specifically, given an exploration tarpit region, we identify the UI element that the tool acts on right before the region begins, and then we use Toller to disable the UI element for Vet-guided runs. Many tool issues can be targeted by this simple approach. For example, if we disable the advertisement banners that lead to Unresponsive UIs in §5.2, tools will simply not run into the undesirable situation, and they can focus on testing other more valuable app functionalities. In some cases when there are multiple entries to the region and existing traces do not reveal all the entries, the aforementioned approach might fail. We mitigate this limitation by monitoring and controlling the testing progress—currently, if we observe any of the most frequently appearing screens from Excessive Local Exploration regions, we restart the AUT in Vet-guided runs.

We implement UI element disablement by relying on Toller to monitor screen changes during testing and dynamically modify UI element properties. When a target screen (i.e., a UI screen containing any target UI element, as determined by UI hierarchy equivalence check) shows up, we pinpoint the target UI element by matching with the path to each UI element from the root UI element. Once we confirm that the target UI element exists on the current

screen, we instruct TOLLER to disable the UI element, which will not respond to any further action on itself. For the edge case where the action is not on any UI element (e.g., pressing the Back button), we restart the target app once we see the corresponding target screen. Note that there is no need to modify the app installation packages given that we manipulate app UIs dynamically.

5.3.2 Methodology of experiments. We aim to measure the testing effectiveness throughout the entire process of applying VET. For each (tool, app) pair, in addition to the three *initial* runs for trace collection, we perform the experiments for three runs in each of these three settings: (1) Disabling UI elements based on rank-1 regions (rank-1 *guided* runs; see §4.1 for our ranking strategy), (2) Disabling UI element based on rank-1, rank-2, and rank-3 regions (rank-1/2/3 guided runs), and (3) Keep the same settings as initial runs (*comparison* runs). Note that each run also lasts for one hour, and all experiments are conducted in the same hardware and software environment regardless of different settings.

We measure method coverage (numbers of uniquely covered methods in app bytecode) as one testing effectiveness metric in our experiments. Note that methods involved by app initialization (i.e., before tools start to test) are excluded for a more precise comparison of code coverage gain. Upon each (tool, app) pair, we use the following Test Groups (TGs) for effectiveness comparison:

- *TG-1*: three initial runs and three comparison runs.
- *TG-2*: three initial runs and three rank-1 guided runs.
- *TG-3*: three initial runs and three rank-1/2/3 guided runs.

Each group consists of six one-hour runs, intended for reducing random biases. We accumulate the method coverage of all runs within a group for the group's method coverage. Test Group 1 serves as the baseline, while the other two test groups aim to measure how much testing effectiveness gain can VET users expect. The main reason for experimenting with exploration tarpit regions of different ranks is that there can be multiple tool issues on an AUT, and addressing only one of the issues might not suffice.

We also measure the crash triggering capabilities with cumulative numbers of distinct crashes. We consider only crashes from bytecode given that Android apps are predominantly written in JVM languages (Java and Kotlin). Crashes are identified by hashing the code locations in stack traces. We additionally leverage TOLLER to disable each app's `UncaughtExceptionHandler`, which is widely used by industrial apps to collect crash reports and might prevent crash information from being exposed to the Android log system (i.e., Logcat [15]) and captured by our scripts.

It should also be noted that TOLLER monitors, captures, and manipulates UIs with negligible overheads; thus, the testing effectiveness of original runs should remain comparable with and without TOLLER in use. In addition, VET analyzes traces very efficiently, usually requiring only a few seconds on a single trace, while analysis of multiple traces can be trivially parallelized.

5.3.3 Results. Tables 4 and 5 show the effectiveness improvements by comparing three test groups. As can be seen from the results, automatically applying fixes based on VET's identified exploration tarpit regions helps Ape, Monkey, and WCTester achieve up to 4.4%, 15.3%, and 11.3% cumulative code coverage improvements relatively on 16 apps using the same amount of time. Additionally, VET helps Ape, Monkey, and WCTester achieve up to 2.1x, 2.1x, and 1.9x overall distinct crashes, respectively. It should be noted that VET's automatic approach does not address all the tool issues—some issues, especially those rooting in tool implementations, are likely addressable by only humans.

For most (tool, app) pairs with improvements, considering only rank-1 exploration tarpit regions is sufficient for code coverage gain. However, there are cases where code coverage increases considerably when we consider rank top-3 regions instead. One explanation is that there are multiple applicability issues, or multiple instances of exploration tarpit corresponding to the same applicability issue. For example, when applying Monkey on the app Nike Run Club, there are multiple ways to enter a hard-to-escape functionality as depicted by the motivating example in §2. If we block only one entry, Monkey can still find other ways to enter the functionality (despite being more difficult) and waste time there.

There are also cases where code coverage decreases when we consider rank-3 exploration tarpit regions. One reason is that lower-ranked regions might not capture real issues, but VET tries to "fix" them anyway, indeliberately interfering with normal functionalities. In the case of applying WCTester on Spotify, the rank-3 region does not reveal any tool issue, according to our observation. "Fixing" this region can cause VET to restart the app when one major functionality shows up. Consequently, WCTester is unable to explore that functionality to achieve more coverage in guided runs.

5.4 RQ3. Missed Tool Issues

We show our analysis of tool issues that are not revealed by any exploration tarpit regions (i.e., false negatives) reported by VET.

5.4.1 Methodology. We propose using issue-specific detection tools to discover hidden tool issues (in all the collected traces), which can provide an estimation of how likely any issue is missed. Specifically, we summarize the characteristics of two issues from §5.2, *App logout* and *Unresponsive UIs*, to design two approaches specifically targeting these two issues. The reason for choosing the aforementioned issues is that (1) they have a substantial appearance among all issues that we have identified, and (2) their existence is relatively more straightforward to be determined using our infrastructure. We do not adopt manual inspection due to the subjectivity and the error-prone nature of manual judgments, especially given that we need to look at all the collected traces entirely.

Our issue-specific detection approaches work as follows:

- *App logout.* We first manually look into the activity list of each app with 'Login' ticked in Table 2 and identify the subset of activities that are used for logging in to the app. Then, when we analyze a given trace, we find the first and the last occurrence of any activity that belongs to the aforementioned list. If there is any occurrence, and the time distance between the first and the last occurrence is at least t_{min}, we regard that an issue of App logout is found in the trace.
- *Unresponsive UIs.* In our investigation, we find only one case that leads to Unresponsive UIs: when an advertisement banner is clicked. Consequently, a new activity can be observed, where the activity belongs to the Google AdMob SDK and has the same activity ID across different apps. Thus, we simply look for continuous appearance (i.e., there is no other activity in between)

Table 4: Cumulative code coverage statistics.

App Name	Ape					Monkey					WCTester				
	$\#M_1$	Rank-1		Rank-1/2/3		$\#M_1$	Rank-1		Rank-1/2/3		$\#M_1$	Rank-1		Rank-1/2/3	
		$\#M_2$	ΔM_2	$\#M_3$	ΔM_3		$\#M_2$	ΔM_2	$\#M_3$	ΔM_3		$\#M_2$	ΔM_2	$\#M_3$	ΔM_3
AccuWeather	21977	22485	2.3%	22538	2.6%	14830	29266	97.3%	24990	68.5%	14982	15104	0.8%	14797	-1.2%
AutoScout24	17245	17274	0.2%	22713	31.7%	23455	23270	-0.8%	23085	-1.6%	18637	22157	18.9%	22154	18.9%
Duolingo	15186	15299	0.7%	15510	2.1%	13676	14598	6.7%	14582	6.6%	12319	14625	18.7%	14450	17.3%
Flipboard	9594	9742	1.5%	9510	-0.9%	5949	7482	25.8%	7125	19.8%	7872	7901	0.4%	8265	5.0%
Merriam-Webster	9056	9093	0.4%	9093	0.4%	5617	8992	60.1%	9275	65.1%	9328	9089	-2.6%	9010	-3.4%
Nike Run Club	30523	29937	-1.9%	29939	-1.9%	24472	24592	0.5%	26645	8.9%	19754	21936	11.0%	22460	13.7%
OneNote	6681	7134	6.8%	7028	5.2%	7131	7114	-0.2%	7381	3.5%	6453	6675	3.4%	6933	7.4%
Quizlet	17000	17114	0.7%	16900	-0.6%	13722	13995	2.0%	13995	2.0%	14679	14865	1.3%	14448	-1.6%
Spotify	19759	21475	8.7%	21475	8.7%	20616	22200	7.7%	21486	4.2%	18897	29298	55.0%	19632	3.9%
TripAdvisor	29857	30645	2.6%	31006	3.8%	16773	20919	24.7%	20919	24.7%	26773	28467	6.3%	28180	5.3%
trivago	20706	20710	0.0%	20711	0.0%	20216	20489	1.4%	20482	1.3%	19952	19964	0.1%	20032	0.4%
Wattpad	22960	22668	-1.3%	22447	-2.2%	14541	13717	-5.7%	15276	5.1%	15067	15884	5.4%	15982	6.1%
WEBTOON	32933	31674	-3.8%	31599	-4.1%	31477	30176	-4.1%	30176	-4.1%	25720	27659	7.5%	27659	7.5%
Wish	8829	8850	0.2%	9106	3.1%	8490	8522	0.4%	8269	-2.6%	6948	7191	3.5%	7207	3.7%
Youtube	26874	33301	23.9%	33757	25.6%	29316	32087	9.5%	35892	22.4%	22179	29143	31.4%	30233	36.3%
Zedge	42899	43074	0.4%	43433	1.2%	31245	38103	21.9%	44931	43.8%	36671	37464	2.2%	38343	4.6%
Average	**20755**	**21280**	**2.5%**	**21673**	**4.4%**	**17595**	**19720**	**12.1%**	**20282**	**15.3%**	**17264**	**19214**	**11.3%**	**18737**	**8.5%**

Notes: '$\#M_n$' shows the total number of covered methods in Test Group n. $\Delta M_n = (\#M_n - \#M_1) \div \#M_1 \times 100\%$.

Table 5: Distinct crash statistics.

App Name	Ape			Monkey			WCTester		
	$\#C_1$	$\#C_2$	$\#C_3$	$\#C_1$	$\#C_2$	$\#C_3$	$\#C_1$	$\#C_2$	$\#C_3$
AccuWeather	2	4	9	0	5	4	0	2	4
AutoScout24	1	1	1	2	1	1	0	0	0
Duolingo	1	2	2	0	0	0	1	1	1
Flipboard	0	0	1	0	0	1	1	1	0
Merriam-Webster	2	3	3	0	5	7	0	0	0
Nike Run Club	1	1	1	6	3	5	0	0	1
OneNote	0	2	5	0	2	1	0	1	0
Quizlet	0	1	0	0	0	0	1	1	1
Spotify	1	1	1	0	0	0	0	0	0
TripAdvisor	3	5	5	0	1	1	1	1	1
trivago	1	1	2	0	1	2	2	1	2
Wattpad	2	2	2	0	0	0	2	3	2
WEBTOON	1	1	1	1	0	0	0	3	3
Wish	2	4	2	2	1	1	0	0	0
Youtube	0	0	0	0	0	0	1	1	2
Zedge	0	0	0	0	0	0	0	0	0
Total	**17**	**28**	**35**	**11**	**19**	**23**	**9**	**15**	**17**

Notes: '$\#C_n$' is for total # triggered unique crashes in Test Group n.

of the aforementioned activity ID on the given trace. Note that we require the appearance to be continuous so as to exclude the cases where the tool (such as Ape) chooses to restart the app. If the appearance lasts for at least t_{min}, we regard that an issue of Unresponsive UIs is found in the trace.

In order for a detected issue to be considered covered by our general-purpose algorithms, we require that at least one algorithm-identified exploration tarpit region covers at least 1/2 of the time length within which the detected issue appears.

5.4.2 Results. We apply the specialized approaches on all 144 collected traces. As a sanity check, we find that all the manually-discovered App logout and Unresponsive UIs issues are covered by the specialized approaches. We compare the results against identified regions from Vet's general-purpose algorithms and perform manual confirmation. We find only several cases where the Vet algorithms do not yield accurate results, discussed as follows:

- On one trace from applying Monkey on Zedge, Vet misses one Unresponsive UIs issue by not reporting any covered region. We find that the Excessive Local Exploration algorithm prioritizes another region over any region covering this issue. However, we find that this issue is also present in other traces from the same (tool, app) pair, and Vet identifies and addresses this issue.
- On each of two other traces from Ape on Duolingo and Monkey on Zedge, there are two instances of the same Unresponsive UIs issue, and Vet reports only one of them. The inaccuracy is also caused by the prioritization strategy of the Excessive Local Exploration algorithm. However, since two instances point to the same issue on both traces, Vet is still effective.
- On one trace from Ape on Quizlet, Ape logs out about only 5 minutes after testing starts, but Vet reports only 22 minutes of exploration tarpit. We find Quizlet's UI design to be somewhat unique: the app has a special entry to some of the main functionalities in its landing page that is accessible without logging in. The entry is buried within a paragraph of texts, and the texts are shown only after a specific combination of swiping. Ape is able to find this special entry in this run, making Vet confused. Nevertheless, Vet is still able to find the correct trigger action from other traces.

6 DISCUSSION AND LIMITATION

We are mainly focusing on making Vet useful in the context of automated UI testing. However, it should be noted that Vet's potential usage scenarios are beyond automated UI testing. One usage case is for app UI quality assurance, where an app might have UI design issues with one or more functionalities. As a result, human users may face difficulties locating their desired features. When a human user runs into such situations, he/she will then likely search for the desired features through repeated (and ineffective) exploration around a few functionalities, and the difficulties can be reflected by the collected user behavior statistics. By subsequently utilizing Vet's identification of exploration tarpits, we can quickly know which functionalities likely have the aforementioned UI design issues, potentially from numerous traces collected from end-users, and address these issues in a more timely manner.

One question is whether VET is capable of differentiating testing scenarios (1) that a tool is supposed to handle but does not (i.e., tool issues), and (2) that a tool is not expected to handle (i.e., beyond tool capabilities, such as apps requiring special inputs). We would like to point out that it is inherently difficult to differentiate these two types of scenarios due to lacking specifications over tool capabilities. On the other hand, VET can help users identify (and mitigate) the cases beyond a tool's capabilities, being already useful.

We acknowledge that TOLLER, the utility that monitors, captures, and manipulates UIs for VET, still has limitations. For example, the current implementation of TOLLER does not capture text inputs. However, adding support for text capturing is achievable with engineering efforts. Moreover, TOLLER's limitations do not prevent VET from being generalizable.

7 THREATS OF VALIDITY

A major external threat to the validity of our work is the environmental dependencies of our subject apps. More specifically, many of the industrial apps in our experiments require networking for main functionalities to be usable, and it is possible for such dependency to change the behaviors of these apps despite our efforts to make our experiment environment consistent across different runs. In order to reduce the influences of environmental dependencies in our experiments, we repeat each experiment setting by three times and use aggregated metrics in our paper. We additionally control each tool's internal randomness by setting a constant random seed for each app. Nevertheless, this threat can be further reduced by involving more repetitions in our experiments.

A major internal threat to the validity of our work comes from the manual analysis of collected traces. We need to manually determine whether the exploration tarpit regions reported by VET indeed reveal any tool issue. Consequently, related evaluation results can be influenced by subjective judgments. However, it should be noted that any work involving manual judgments in the evaluation is vulnerable to this threat.

8 RELATED WORK

Automated UI testing for Android. There have been various tools over years of development. The earliest efforts include Monkey [12], a randomized tool that does not consider app UIs or coverage information. The superior efficiency brings strong competitiveness to the simple tool. Subsequent efforts result in tools mainly driven by randomness/evolution [24, 26, 44], UI modeling [7, 17, 18, 22, 35], and systematic exploration [1, 2, 25]. Recent work [10] proposes time-travel testing (referred to as TTT) to help Android UI testing escape from ineffective AUT states including loops and dead ends. TTT uses checkpoint and restore—checkpointing progressive states and restoring those states after loops and dead ends are detected.

VET is different from TTT in terms of design goals. First, TTT aims to recover from ineffective exploration, while VET mainly focuses on prevention. Second, exploration tarpits in VET are more general than TTT's lack-of-progress definitions. One example is logging out, where tools assisted by VET can still explore a fraction of app functionalities, such as registering and resetting passwords. Loops and dead ends are not necessarily present when exploring an app with only a few functionalities. Third, VET aims to enhance existing testing tools without the need to understand their internal design or implementation, instead of building a new testing tool that excels at all apps.

The design of VET brings a few advantages. First, as acknowledged by TTT [10], the state recovery may lead to inconsistent app states when testing apps with external state dependencies that are maintained at the server side. Note that controlling server-side states is challenging, e.g., many industrial apps use external services that the apps have no control of. VET's preventive strategy avoids this limitation. Second, VET's preventive strategy does not incur overhead for lack-of-progress detection or state recovery in guided runs. This strategy is specifically useful when a tool repeatedly gets into exploration tarpits. Third, VET does not require additional device support for state recovery (such as RAM data restoring).

Trace Analysis. Our work is related to log and trace analysis. Existing work has been focusing on analyzing the logs generated by program code through techniques including anomaly detection [11, 28, 47, 52], cause analysis [6, 9, 32, 49, 54], failure reproduction [48], and performance-issue detection [50, 51]. Our work focuses on UI traces with the goal of understanding UI exploration tarpits, which are different from logs produced by program code.

Parallel Testing. Our work is also related to parallel testing in the sense of producing multiple variants of the target program (i.e., customizing the AUT by manipulating the UI entries) for the testing tool to work on. Related work on parallel testing includes parallelizing mutation testing [27], symbolic execution [5, 34], and the debugging process [20].

9 CONCLUSION

We have exploited the opportunities of improving Android UI testing via automatically identifying and addressing exploration tarpits. Specifically, we have presented VET, a general approach and supporting system for effectively identifying and addressing exploration tarpits. We have designed specialized algorithms to support VET's concepts. Our evaluation results have shown that VET identifies exploration tarpits that cost up to 98.6% of testing time budget, revealing various issues hindering testing efficacy. By trying to automatically fix the discovered issues, VET helps the Android UI testing tools under evaluation with achieving up to 15.3% higher code coverage relatively and triggering up to 2.1x distinct crashes.

ACKNOWLEDGMENTS

This work was partially supported by 3M Foundation Fellowship, NSF grants (CNS-1564274, SHF-1816615, CNS-1956007, and CCF-2029049), a Facebook Distributed Systems Research award, Microsoft Azure credits, and Google Cloud credits.

REFERENCES

[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *FSE*.
[2] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *OOPSLA*.
[3] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria. In *ASE*.
[4] Frederick P. Brooks. 1995. *The Mythical Man-Month (Anniversary Ed.).* Addison-Wesley Longman Publishing Co., Inc.

[5] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*.

[6] An Ran Chen. 2019. An Empirical Study on Leveraging Logs for Debugging Production Failures. In *ICSE*.

[7] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *OOPSLA*.

[8] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *ASE*.

[9] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *OSDI*.

[10] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-Travel Testing of Android Apps. In *ICSE*.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *CCS*.

[12] Google. 2021. Android Monkey. https://developer.android.com/studio/test/monkey

[13] Google. 2021. Android View. https://developer.android.com/reference/android/view/View

[14] Google. 2021. Android ViewGroup. https://developer.android.com/reference/android/view/ViewGroup

[15] Google. 2021. Logcat command-line tool. https://developer.android.com/studio/command-line/logcat

[16] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. 2017. AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications. In *ICSME*.

[17] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications via Model Abstraction and Refinement. In *ICSE*.

[18] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *MobiSys*.

[19] IDC and Gartner. 2019. 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/

[20] James A. Jones, James F. Bowring, and Mary Jean Harrold. 2007. Debugging in Parallel. In *ISSTA*.

[21] Mobile Labs. 2018. 10 Best Android & iOS Automation App Testing Tools. https://mobilelabsinc.com/blog/top-10-automated-testing-tools-for-mobile-app-testing

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *ICSE-C*.

[23] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *UIST*.

[24] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *ESEC/FSE*.

[25] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *FSE*.

[26] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *ISSTA*.

[27] Pedro Reales Mateo and Macario Polo Usaola. 2013. Parallel mutation testing. *Software Testing, Verification and Reliability* (2013).

[28] Cristina Monni and Mauro Pezzè. 2019. Energy-Based Anomaly Detection a New Perspective for Predicting Software Failures. In *ICSE-NIER*.

[29] MoQuality. 2021. How to Automate Mobile App Testing. https://www.moquality.com/blog/How-to-Automate-Mobile-App-Testing

[30] Šarūnas Packevičius, Dominykas Barisas, Andrej Ušaniov, Evaldas Guogis, and Eduardas Bareiša. 2018. Text Semantics and Layout Defects Detection in Android Apps Using Dynamic Execution and Screenshot Analysis. In *ICIST*.

[31] Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc.

[32] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. 2013. Assisting Developers of Big Data Analytics Applications When Deploying on Hadoop Clouds. In *ICSE*.

[33] SmartBear. 2021. Mastering the Art of Mobile Testing. https://smartbear.com/resources/ebooks/mastering-the-art-of-mobile-testing/

[34] Matt Staats and Corina Pundefinedsundefinedreanu. 2010. Parallel Symbolic Execution for Structural Test Generation. In *ISSTA*.

[35] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *ESEC/FSE*.

[36] Testsigma. 2021. How to perform Mobile Automation Testing of the UI? https://testsigma.com/blog/how-to-perform-mobile-automation-testing-of-the-ui/

[37] VET Artifacts. 2021. https://github.com/VET-UI-Testing/main.

[38] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools. In *ISSTA*.

[39] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *ASE*.

[40] Laurence A Wolsey and George L Nemhauser. 1999. *Integer and combinatorial optimization*. John Wiley & Sons.

[41] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, and Jian Lu. 2019. DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps. In *CCS*.

[42] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. IconIntent: Automatic Identification of Sensitive UI Widgets Based on Icon Classification for Android Apps. In *ICSE*.

[43] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *FASE*.

[44] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *MoMM*.

[45] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?. In *FSE*.

[46] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* (12 1989).

[47] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. 2019. Robust Log-Based Anomaly Detection on Unstable Log Data. In *ESEC/FSE*.

[48] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *SOSP*.

[49] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *SOSP*.

[50] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *OSDI*.

[51] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. Lprof: A Non-Intrusive Request Flow Profiler for Distributed Systems. In *OSDI*.

[52] Zilong Zhao, Sophie Cerf, Robert Birke, Bogdan Robu, Sara Bouchenak, Sonia Ben Mokhtar, and Lydia Y Chen. 2019. Robust Anomaly Detection on Unreliable Data. In *DSN*.

[53] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *ICSE-SEIP*.

[54] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs. In *ESEC/FSE*.