TextExerciser: Feedback-driven Text Input Exercising for Android Applications

Yuyu He*,1, Lei Zhang*,1, Zhemin Yang[1], Yinzhi Cao[2], Keke Lian[1], Shuai Li[1], Wei Yang[3], Zhibo Zhang[1]
Min Yang[1], Yuan Zhang[1], Haixin Duan[4]
1: Fudan University, 2: Johns Hopkins University, 3: University of Texas at Dalla, 4: Tsinghua University
1: {heyy16, lei_zhang14, yangzhemin, kklian18, lis19, zbzhang15, m_yang, yuanxzhang}@fudan.edu.cn
2: yinzhi.cao@jhu.edu, 3: wei.yang@utdallas.edu, 4: duanhx@tsinghua.edu.cn
**: The first two authors have contributed equally to this work.*

***Abstract*—Dynamic analysis of Android apps is often used together with an exerciser to increase its code coverage. One big obstacle in designing such Android app exercisers comes from the existence of text-based inputs, which are often constrained by the nature of the input field, such as the length and character restrictions.**

In this paper, we propose `TextExerciser`, an iterative, feedback-driven text input exerciser, which generates text inputs for Android apps. Our key insight is that Android apps often provide feedback, called hints, for malformed inputs so that our system can utilize such hints to improve the input generation.

We implemented a prototype of `TextExerciser` and evaluated it by comparing `TextExerciser` with state-of-the-art exercisers, such as The Monkey and DroidBot. Our evaluation shows that `TextExerciser` can achieve significantly higher code coverage and trigger more sensitive behaviors than these tools. We also combine `TextExerciser` with dynamic analysis tools and show they are able to detect more privacy leaks and vulnerabilities with `TextExerciser` than with existing exercisers. Particularly, existing tools, under the help of `TextExerciser`, find several new vulnerabilities, such as one user credential leak in a popular social app with more than 10,000,000 downloads.

***Index Terms*—Dynamic Analysis, Android Security, Text Input Generation, Android Application Testing**

I. Introduction

Dynamic analysis is widely used in the past to analyze Android apps for vulnerabilities [1]–[3], malicious behaviors [4]–[8], and privacy leaks [9]–[13]. One important component, often used together with dynamic analysis, is an application or UI exerciser that drives Android apps to reach different code branches so that the analysis can be performed completely with a high code coverage. Examples of such exercisers are like the most famous fuzzing tool, The Monkey [14], which randomly generates UI events for Android apps. Some other works [5], [9], [10], [15]–[17] also follow up on The Monkey (Monkey for short) to exercise apps more thoroughly with even higher code coverage.

Although these exercisers can successfully drive Android apps, one critical obstacle is that many apps require text-based inputs with super-linear possibilities that existing exercisers cannot enumerate in a reasonable amount of time. Furthermore, these inputs often require a non-trivial constraint that is hard to be satisfied during analysis. For example, a personal profile description field of an Android app may require texts that range between 8 and 1,600 characters and do not contain any special characters. Such constraints for text-based inputs are very popular in Android apps: Our manual inspection shows that text inputs in 150 out of top 200 free non-game Android apps have at least one constraint.

Due to the difficulties of generating text inputs in exercising Android apps, some researchers propose to adopt either a heuristic approach or predefined information. For example, AppsPlayground [18] and Arnatovich et al. [19] summarize all the input patterns for a specific field, such as username and password. Liu et al. [20] rely on machine learning to automatically learn the input patterns. Several other tools [2] use predefined third-party login such as Facebook login and Google sign-in to circumvent text input UI such as login page. However, these input patterns are diversified: 130 out of our previously-studied 200 apps have a unique way to constrain text inputs, which either do not support third-party login or require unique input constraints that are different from general heuristics rules. Furthermore, these text fields are often correlated, e.g., the value of a "maximum wage" field should be larger than the one of a "minimum wage". Therefore, existing works, such as rule or learning-based summarization of text input patterns, often fail to satisfy these unique constraints—but a single failure will stop the entire exercising of the target app.

In this paper, we propose `TextExerciser`, an iterative, feedback-driven text input exerciser, which generates inputs for text fields of Android apps. The *key* insight here is that if a text input does not satisfy the enforced constraints, the Android app—either the client-side program or the server-side validation—will provide clues or hints for the malformed input, which can be used as a feedback for improvement. Let us look at a concrete example: Say we input a password with five letters into an Android app with a text field that requires a length of at least six characters. The Android app will prompt a hint saying that "the password must be at least 6 characters" so that the user knows how to proceed—and at the same time, such a hint can be used to refine our text input generation.

Specifically, here is how `TextExerciser` works iteratively to generate text inputs. `TextExerciser` first extracts all the hints related to malformed text inputs based on information that appears after the inputs are fed to the app. Then, `TextExerciser` adopts natural language processing to parse the extracted hints into a syntax tree and understand the semantics. Next, `TextExerciser` generates constraints

DOI 10.1109/SP40000.2020.00071

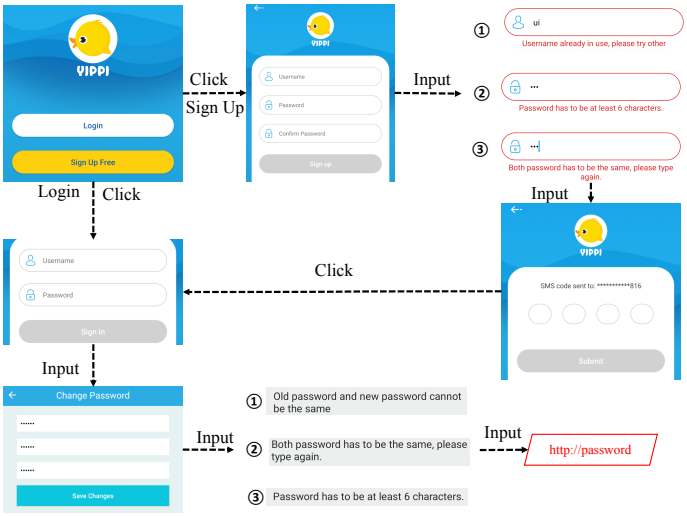


Figure 1. A Motivating Example of `TextExerciser` (The Yippi app contains a vulnerability that transfers password in an HTTP protocol, which can be sniffed by a man-in-the-middle, only on the "Change Password" interface. A dynamic analyzer requires many text-based inputs in order to reach the vulnerable location and find the vulnerability.)

according to hint semantics and outputs a possible input using a constraint solver. Lastly, `TextExerciser` feeds the input back to the target app—if the input still cannot satisfy the constraints, `TextExerciser` will iterate the process until a valid input is found.

We evaluate `TextExerciser` on 6,000 popular apps collected from Google Play. The results show that `TextExerciser` achieves higher code coverage than state-of-the-art approaches like Monkey [14], Stoat [21] and DroidBot [22]. We also combine `TextExerciser` with dynamic analysis tools, such as TaintDroid [11] and ReCon [23]. Our evaluation results show that tools with `TextExerciser` can find more privacy leaks and vulnerabilities, such as a previously-unknown user credential leakage vulnerability in a popular social app, called Coco, with more than 10,000,000 downloads on Google Play. We have responsibly reported all the vulnerabilities to app developers—the developers of Coco have fixed the vulnerability internally and will release a new version shortly.

Contributions. The main contributions of our work are summarized as follows:

- We propose the first feedback-driven input exerciser that iteratively generates text inputs using a constraint solver based on hints from the target app.
- We implement a prototype of our text input exerciser and the source code of `TextExerciser` is available at GitHub [24].
- We evaluate the performance of `TextExerciser` on popular Google Play apps. The evaluation result shows that `TextExerciser` achieves higher code coverage than state-of-the-art tools and also finds more privacy leaks and vulnerabilities when combined with existing dynamic analysis tools.

II. A Motivating Example

In this section, we use a real-world example to motivate the use of `TextExerciser` to exercise Android apps. The example, called Yippi as shown in Figure 1, is a message app with 100,000+ downloads, which, under the developers' descriptions, allows user communications with a focus on entertainment and "security".

The Yippi app has an vulnerability of user-credential leakage that requires heavy text-based exercising. We responsibly reported this vulnerability to the app developer—but have not received any feedback yet. Here are the details. Once an user launch the app for the first time, Yippi will ask for the user to sign up for a new account and then log in with the account. Therefore text inputs are needed to sign-up a new user account and log into the app. After login, at the "change password" page, another text inputs are needed to trigger the password transfer so that existing dynamic analysis tools can find that Yippi is insecure as it transfers changed passwords in the HTTP protocol. Note that all other password transfers in Yippi are done securely via an HTTPS channel.

It is challenging to generate inputs for Yippi, due to several input requirements:

- Username Uniqueness. The inputs to the "username" field need to be unique when comparing with others in the app's database. That is, if one chooses a used username in the database, Yippi will return a warning, saying that "Username already in use, please try other".
- Length Requirement. The inputs to the "password" field need to satisfy certain conditions, i.e., with a length of at least 6 characters: Yippi will also display a hint if the condition is not satisfied.
- Joint-field Dependencies. The inputs to the "confirm password" need to match the one to the "password" field—leading to a joint-field constraint. Yippi will also alert the users if these two fields do no match.
- SMS Authentication. After the initial sign-up page, Yippi asks for a validation code sent via SMS.

We studied existing exercising tools such as Monkey and found that none of them can exercise Yippi and trigger the vulnerability. Monkey will stop at the sign-up page, failing to exercise the app beyond the login wall. Prior exercisers [2], [21], [22], [25], [26] that rely on pre-defined inputs or third-party logins cannot generate valid inputs for Yippi, because the constraints are complex and Yippi does not support any third-party logins. For example, many of these pre-defined usernames in prior works are used before by others in Yippi's database and the pre-defined passwords may also fail to satisfy the specific requirement.

III. Methodology

In this section, we introduce the methodology of exercising text inputs for Android apps. We start from introducing the workflow of `TextExerciser` and then present each phase of `TextExerciser` individually.

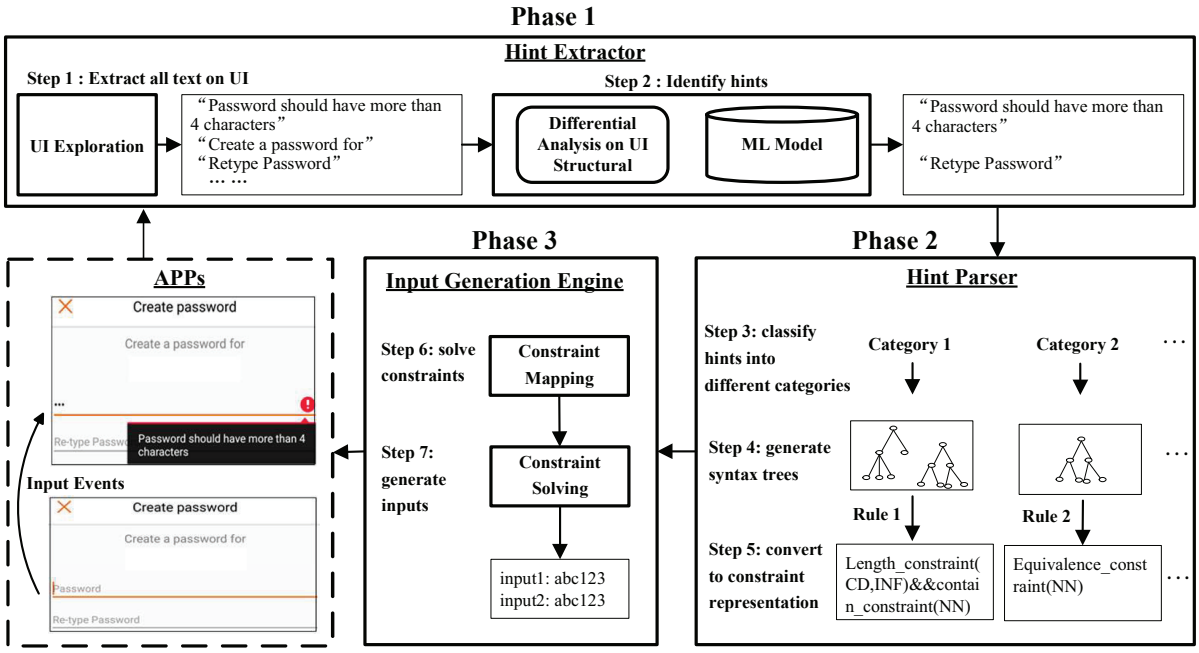


Figure 2. The overall architecture of `TextExerciser`. There are three phases to exercise an app: hint extraction (Phase 1), hint parsing (Phase 2), and input generation (Phase 3). If a generated input fails, `TextExerciser` will repeat these three phases based on newly collected feedback until a valid input is generated.

A. System Workflow

`TextExerciser` is a feedback-driven text exerciser that understands hints shown on user interfaces of Android apps and then extracts corresponding constraints. The high-level idea of understanding these hints is based on an observation that these hints with similar semantics often have a similar syntax structure—and therefore `TextExerciser` can cluster these hints based on their syntax structures and then extract the constraints from the syntax structure. Now, let us give some details of `TextExerciser`'s workflow.

The exercising has three phases, seven steps as shown in Figure 2. First, `TextExerciser` extracts all the texts in the app's UI (Step 1) and then identifies static hints via a learning-based method and dynamic hints via a structure-based differential analysis (Step 2). Second, `TextExerciser` parses all the extracted hints via three steps: classifying hints into different categories (Step 3), generating syntax trees for each hint (Step 4), and interpreting the generated tree into a constraint representation form (Step 5). Lastly, `TextExerciser` generates a concrete input by feeding constraints into a solver (Step 6), e.g., Z3. Then, `TextExerciser` solves the problem, feeds generated inputs back to the target Android app and extracts feedbacks, such as success and another hint (Step 7). In the case of another hint, `TextExerciser` will iterate the entire procedure until `TextExerciser` finds a valid input.

Now let us look at our motivating example in §II again to explain `TextExerciser`'s workflow. We start from the sign-up page, which has three text input fields, i.e., "username", "password" and "confirm password". `TextExerciser` generates a random input to the username field: If the username is used in the database, Yippi returns a "username used" hint. `TextExerciser` will then parse the hint and generate a new username. The "password" and "confirm password" are handled together by `TextExerciser`: based on the hint that "Both password has to be the same"[1], `TextExerciser` will convert the hint into a constraint that the value of both fields need to be the same and then generate corresponding inputs.

After `TextExerciser` generates inputs for the first sign-up page, Yippi asks the user to input a code that is sent to a phone number. `TextExerciser` will first extract hints related to the phone number page, understand that this is a phone number, and then input a pre-registered phone number to the field. Next, `TextExerciser` will automatically extract the code from the SMS and solve the constraints by inputting the code to Yippi.

In order to find the aforementioned vulnerability in §II, `TextExerciser` also generates text inputs to the "Change Password" page. Particularly, `TextExerciser` extracts the password matching hint and another hint that distinguishes old and new passwords, converts them into constraints and then generates corresponding inputs so that existing dynamic analysis tools can find the vulnerability.

B. Hint Extraction

The first phase of `TextExerciser` is to extract hints related to text inputs from an Android app. There are two types of available hints in Android apps, i.e., dynamic and static. A dynamic hint appears once a user inputs an incorrect text into the Android app, e.g., the app may alert the user that a specific username has been registered by others. As a comparison, a static hint appears together with the text input

[1]The sentence with a grammar error is from the Yippi app.

field, e.g., the app may state that a password should contain a special character.

TextExerciser extracts dynamic hints via a differential analysis that compares widgets before and after inputting a text into the app. Information that appears in the widget after text input is considered as a hint. An example is shown in Figure 3.(a): After the user inputs a short description into her profile, a hint appears and alerts the user that the description should be at least 20 characters long. Such differential information could also appear in the form of a popup window, such as examples shown in Figure 3.(b).

TextExerciser then extracts static hints via a learning-based approach. Specifically, we train a neural network model for classification. The positive training samples come from dynamic hints extracted from these Android apps via differential analysis as a training set; the negative samples come from information extracted from app windows without any text input, i.e., those that presumably are not hints to text inputs.

Next, TextExerciser needs to map extracted hints to corresponding input fields. TextExerciser adopts two methods in the mapping. First, TextExerciser maps keywords extracted from a hint to the text related to the input field. For example, if both the hint and the input field mention "password" as shown in Figure 4.(a), TextExerciser considers an input generated following this hint is for the corresponding input field. Second, TextExerciser adopts a shortest-distance method to find the closest input field. Note that the distance definition as illustrated in Figure 4.(b) is the relative location of widgets instead of Euclidean distance because the widget size depends on the nature of the input. Note that multiple hints may be mapped to a single input field, because an input field may have more than one requirement.

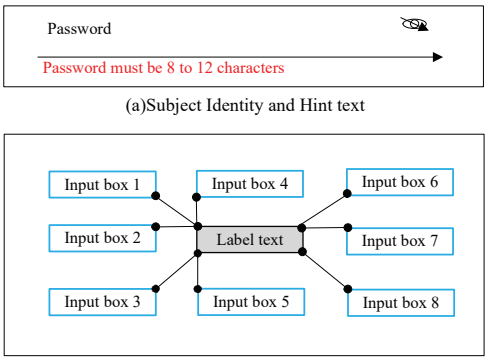
(a) Differential Widget Example (b) Popup Notification Example

Figure 3. Example for hint extraction

C. Hint Parsing

In this phase, TextExerciser classifies extracted hints, parses them into syntax trees, and then generates constraints in a representation acceptable by solvers.

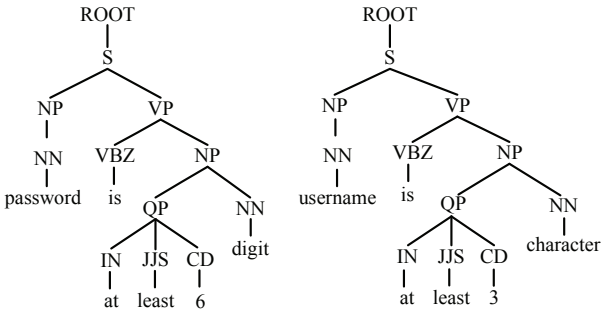
1) Hint Classification: TextExerciser classifies the extracted hints into pre-defined categories using a multi-class model. Now, we first describe how to pre-define hint categories and then present the classification procedure. We categorize hints based on their semantics, i.e., how they enforce restrictions on text inputs, by manually surveying top 1,200 free non-game apps from Google Play. The pre-defined categories have



(a)Subject Identity and Hint text

(b)Distance Calculation Between Hint Text and Input Field

Figure 4. Hints to Input Field Mapping.



(a) Syntax Tree A (b) Syntax Tree B

Figure 5. Example of Syntax Tree.

4 major, 10 minor and 18 sub-minor as shown in Table I—we now describe these major hint categories below.

- Precise Single-field. This category refers to that a hint precisely describes the requirement, such as the length of a text input.
- Fuzzy Single-field. This category refers to that a hint vaguely describes the requirement, e.g., the length is too small or the input contains invalid characters.
- Precise Joint-fields. A hint in this category indicates that two input fields are correlated, e.g., the value of the maximum salary field should be larger than the value of the minimum.
- Fuzzy Joint-fields. A hint in this category indicates a vague correlation between two fields, e.g., the length of a phone number depends on another country field.

Each major category has several minor categories based on the constraint type, such as length and value, which can be further divided into sub-minors. We manually label all, i.e., 1,548 of, the hints from these apps with all the sub-minor categories and then train a multi-class classifier using both CNN and RNN [27]. TextExerciser adopts this classifier to determine the categories of a given hint extracted from Android apps.

2) Syntax Tree Generation: In this step, TextExerciser pre-processes the extracted hints and then generates a syntax tree using the Stanford parser [28]. The pre-processing has

Table I
HINT CATEGORIES SUMMARIZED FROM TOP FREE GOOGLE PLAY APPS. WE MANUALLY TEST ALL THE TEXT FIELDS THAT A HUMAN CAN FIND AND THEN EXTRACT CORRESPONDING HINTS. NOTE THAT, "#A": THE NUMBER OF HINTS IN THIS CATEGORY, AND "#U": THE NUMBER OF UNIQUE HINTS IN THIS CATEGORY.

MajorCategory	MinorCategory	SubMinorCategory	Id	Example	#A	#U
Precise Single-field	Length Constraints	The lower bound of input length	C1	Your password must be at least 6 characters long	295	6
		The upper bound of input length	C2	Your password should be shorter than 6 characters	15	3
		A range of input length	C3	Your User ID must be between 6 to 62 characters	87	4
		A fixed input length	C4	The PIN must have 6 characters	71	8
	Existence Constraints	Input should contain certain characters	C5	The code needs to contain numbers	47	6
		Input should not contain certain characters	C6	Don't use a whitespace in your username	13	4
	Value Constraints	The lower bound of value	C7	Expiration date must be at least 30 days from today	19	4
		The upper bound of value	C8	Child must be less than 18 years old	3	2
		A range of value	C9	Specify your weight between 10 and 999	5	2
Fuzzy Single-field	Length Constraints	Require longer input	C10	Nickname is too short	7	2
		Require shorter input	C11	Your entry for the about field is too long	6	2
	Value Constraints	Require larger value	C12	Date is too small	2	1
		Require smaller value	C13	The amount which you have specified exceeds your limits	1	1
	Non-directional Constraints	Non-directional Constraints on invalid input	C14	Email address format is invalid.	806	5
Precise Joint-fields	Equivalence Constraints	The equivalence of two input fields	C15	New password does not match	134	3
	Non-repetitive Constraints	Value of two input fields can't the same	C16	This username is already taken	30	2
	Value Restriction	The comparison of values in two input fields	C17	Chosen minimum salary higher than chosen maximum salary	1	1
Fuzzy Joint-fields	Non-directional Constraints	The relationship of the two field need domain knowledge	C18	Mobile you typed isn't valid for this country	6	1

two steps: (i) redundant sentence removal, and (ii) word normalization. First, `TextExerciser` divides hints into sentences and removes redundant ones, which are unrelated to constraints, via our hint classifier trained in §III-B. An example is like "Oops! Password must be between 7 and 15 characters. Please try again."—both "Oops!" and "Please try again." are removed in this stage. Second, `TextExerciser` normalizes words, such as replacing spelled-out numbers with corresponding digits and plural words into their corresponding singular one. After pre-processing, `TextExerciser` calls the Stanford parser to generate a syntax tree for each sentence in the extracted hints.

3) Constraint Representation: In this step, `TextExerciser` accepts a syntax tree and a hint category and then generates constraints for the hint. Specifically, `TextExerciser` follows a rule selected based on the hint category to traverse the syntax tree for constraint generation. A traversal rule is hence defined as a query to the syntax tree with three predicates, where the $Select$ predicate specifies the outputted nodes, $Match$ the condition of the selection, and $Generate$ a constraint equation that can be fed into the solver:

$Select$ Node1, Node2, ...

$Match$ Condition1 and Condition2 and ...

$Generate$ Constraint

Let us start with a concrete example. Say, the traversal rule belongs to the length constraint category, i.e., specifying that the length of a text field needs to be larger than a threshold. The traverse needs to find a number as the threshold and a subject, e.g., password and date. Therefore, our traversal rule will look like the following:

$Select$ cd1, np1

$Match$ Follow(qp1=QP,NN) and Contain(qp1,cd1=CD) and First(np1=NP)

$Generate$ LengthConstraint(Subject(np1), Range(cd1, infinity))

Particularly, the traversal rule returns a Cardinal Number Node (CD) $cd1$ as the threshold, and a Noun Phrase (NP) $np1$ as the text field subject, such that $np1$ matches the first (i.e., satisfying the $First$ condition) NP node and $cd1$ is a child node (i.e., satisfying the $Contain$ condition) of a Quantifier Phrase (QP), which is a sibling (i.e., satisfying the $Follow$ condition) of a Noun Node. For instance, if we apply this rule to the syntax tree in Figure 5.(a), $np1$ equals to "password" and $cd1$ equals to "6"; Figure 5.(b) maps to that $np1$ equals to "username" and $cd1$ "3".

Next, once a traversal, following a rule, returns corresponding nodes, `TextExerciser` will follow the $Generate$ predicate to generate constraints. There are three types of constraints:

- Length Constraint. Such a constraint restricts the length of valid inputs to a certain range. For example, $LengthConstraint(A, Range(CD, Infty))$ represents a constraint that the length of A should fall within the range from the value in CD to the infinite value.
- Content Constraint. Such a constraint restricts the content of valid input to a certain format. For example, $ContentConstraint(A, Format(NN))$ depicts that the input A should fulfill the format determined by NN.
- Value Constraint. Such a constraint restricts the range of input values. It contains a scope that represents the range of valid values.

Our example in Figure 5.(a) is converted to a constraint like $LengthConstraint(password, Range(6, Infty))$; similarly, Figure 5.(b) to $LengthConstraint(username, Range(3, Infty))$.

Note that in practice, we often abbreviate a traversal, e.g., as follows:

$$LowerBound :: Follow(QP, NN)\&\&Contain(QP, CD)\&\&First(NP) \rightarrow LengthConstraint(Subject(NP), Range(CD, Infty)) \&\&ContentConstraint(Subject(NP), Format(NN))$$

Such an abbreviation skips the *Select* predicate as it is embedded as part of the *Match* predicate. The relationship between *Match* and *Generate* is also abbreviated as an inference symbol. We may also skip corresponding variables if the variable is unique. In practise, we write 57 traverse rules based on the unique hints in Table I. We now list examples of the rules adopted by TextExerciser in Table II.

D. Input Generation Engine

In this phase, TextExerciser generates inputs that satisfy all the constraints converted from the extracted hints. The first step is to obtain concrete values for each variable in the constraint representation. There are two major sources: external and other fields. External sources involve emails and text messages, in which TextExerciser will pre-register several email account and phone number to receive such values, such as PIN. Other fields involve other text inputs generated by TextExerciser in the case of joint-field constraint—if the inputs to other text fields are generated, TextExerciser will apply joint-field constraint with that concrete value; otherwise, TextExerciser will generate an input without this constraint and apply the constraint for the other involved input field.

The second step is to solve the constraints: Particularly, TextExerciser adopts Z3StrSolver [29], a popular solver, to generate inputs that satisfy all the constraints. An example code for solving two constraints, i.e., LengthConstraint(Range(lower_bound, upper_bound)) and ContentConstraint(A,Format(NN))), is shown in Figure 6. Lines 5–6 are the length constraint, in which TextExerciser asks the solver to generate an input with the length between lower_bound and upper_bound. Lines 8–13 are the content constraint, in which TextExerciser excludes certain characters, such as the one appeared in the hint. TextExerciser also adopts a special constraint to differentiate the generated input from the old ones at Lines 15–16, because the old inputs have already been rejected by the app. Lastly, TextExerciser asks the solver to generate an input at Lines 18–19.

```
1  from z3 import *
2  solver=Solver()
3  input=String('input')
4  //LengthConstraint(lower_bound, upper_bound)
5  solver.add(Length(input) > lower_bound)
6  solver.add(Length(input) < upper_bound)
7  //ContentConstraint(A, content)
8  for char in TOTAL_LETTER:
9      if char not in content:
10         exclude_char = '\\'+str(hex(char))[1:]
11         if len(exclude_char) == 3:
12             exclude_char = exclude_char.replace('x', '
                   x0')
13         solver.add(Not(Contains(input,exclude_char)))
14 // Different from old inputs
15 for old_input in INPUT_HISTORY:
16     solver.add(Not(input == StringVal(old_input)))
17 //Generate a text input
18 if solver.check()==sat:
19    print(solver.model())
```

Figure 6. An Example Z3StrSolver Code with Three Constraints: (i) Length constraint within a certain range (Lines 5–6), (ii) Value constraint that excludes certain characters (Lines 8–13), and (iii) Equivalence constraint (Lines 15–16).

IV. Implementation

We implemented TextExerciser with about 6,350 lines of Python code. Specifically, our constraint extractor has 1,100 lines of code, constraint parser 1,300 lines of code, and input generation engine 1,900 lines of code. TextExerciser also relies on some existing tools in the implementation. In phase 1 (hint extractor), we use the UiAutomator [30], to explore the widgets on UI screen when dynamically running Android apps. In phase 2 (hint parser), we use Stanford parser [28] to generate syntax trees for hint texts. In phase 3 (input generation engine), we use Z3StrSolver [29] to solve the input constraints.

We now describe two implementation details, i.e., the dataset and model used in phase 2 and the validation code extraction in phase 3. First, we collect 1,548 hints from 1,200 top free apps on Google Play and then ask three students to label them during 14 days, which totals to around 50 hours per student. Each hint has three labels—if a discrepancy happens, these three students will discuss and resolve it. Note that the number of discrepancies is relatively small as the labels are mostly straightforward: There only exists 10 out of 1,548 hints. For example, a hint, "Password is case-sensitive", may be labelled as C5 or C14. In the end, all the students agreed to label it as C14 as it is unclear what characters should be included. After labeling, TextExerciser pre-processed all the hints in the training set via two steps: (i) word normalization and (ii) data balancing. That is, TextExerciser replaces all digits with a special tag "TaggedASCD" with Stanford POSTTager, e.g., "4 digits" changes to "TaggedASCD digit", and then leverages SMOTE [31] to balance the samples. In the end, we trained our static hint identification based on a multi-class text classifier [27] with half of all the labeled data and the rest data is used for model validation and evaluation of tool performance.

Second, our validation code handler, used in phase 3 for solving the constraints, is composed of two parts, (i) an email code extractor on a server and (ii) a code receiver written as an Xposed module [32] on the mobile phone. Our email code extractor keeps pulling emails from a pre-registered email address designated for the validation code purpose and also extracts code using a regular expression that matches all the four or six digits in each email. Then, the extractor sends the code and the email subject to the code receiver, which performs a keyword matching of the email subject and the app that requires a code. If a keyword matching fails, the code receiver will also try all the recently-received, unmatched

Table II
EXAMPLES OF HINT TEXTS AND THE CORRESPONDING INTERPRETATION RULES THAT CAN HANDLE THEM. WE SHOW EACH NODE IN THE SYNTAX TREE WITH ITS TYPE (E.G., QP, NN, ETC.), AND USE SUBJECT(NODE) TO PRESENT AN INPUT BOX WHOSE IDENTITY IS DESCRIBED BY A NODE.

Minor-Category	Hint Text Example (lowercase)	Matched Interpretation Rule
Length Constraint	password is at least 3 character	*LowerBound :: Follow(QP,NN) && Contain(QP,CD) && First(NP) → LengthConstraint(Subject(NP), Range(CD,Infty)) && ContentConstraint(Subject(NP), Format(NN))*
Value Constraint	month must be between 1 and 12	*ScopeBound :: Follow(QP)&&Contain(QP,(cd1=CD,cd2=CD))&& First(NP) → ValueConstraint(Subject(NP), Range(cd1,cd2))*
Length Constraint	zipcode must be 5 digit	*FixLength :: Follow(NP)&&Contain(NP,(CD,NN))&& First(NP) → LengthConstraint(Subject(NP), Range(CD,CD)) && ContentConstraint(Subject(NP), Format(NN))*
Existence Constraint	username can not contain space	*Exclusive :: Follow(RB,VP)&&Contain(VP,NN)) && First(NP) → ContentConstraint(Subject(NP), ! Format(NN))*
Equivalence Constraint	new password does not match	*MultipleEquivalence :: Follow(NP)&&Contain(NP,NN) → ValueConstraint(Subject(NP), Range(Subject(NN), Subject(NN)))*
Vague Length Constraint	nickname is too short	*DirectRestric :: Follow(NP)&&Contain(NP,NN) → LengthConstraint(Subject(NP), Subject(NN+1))*

Table III
OVERVIEW OF STATE-OF-THE-ART OPEN-SOURCE DYNAMIC TESTING TOOLS OF ANDROID APPS

Tool	Need of Instrumentation	Text Input Strategy
Monkey [14]	No	Random
Sapienz [25]	System	Random String from App Resource File
Stoat [21]	No	Random
DroidBot [22]	No	Predefined
A3E-Depth-First [26]	App	Random String
`TextExerciser`	No	Feedback Based Mutation

code for the app that needs a code. Note that the code receiver also accepts and extracts code that is sent to the mobile phone directly as a text message. All other steps for this text scenario are the same as the email one.

V. EVALUATIONS

In this section, we evaluate the performance of `TextExerciser` on real-world Android apps via addressing three main research questions below:

- RQ1: is `TextExerciser` more effective than existing tools in exercising Android apps?
- RQ2: can `TextExerciser` improve existing dynamic analysis of Android apps?
- RQ3: is `TextExerciser` efficient for generating text input for popular Android apps?

A. Comparison with State-of-the-art Testing Tools

In this section, we answer RQ1 *(is `TextExerciser` more effective than existing tools in exercising Android apps?)* by comparing the method and activity coverage achieved by each exerciser. Because `TextExerciser` is a specialized text input exerciser while others are general purpose, we replace the text input generator in each general-purpose exerciser with `TextExerciser` and compare the modified version with the original one for code coverage. According to Wang et al. [33] and as shown in Table III, there exists five open-source tools of exercising Android apps, which are Monkey [14], Sapienz [25], Stoat [21], DroidBot [22], and A3E-Depth-First [26]. Sapienz [25] requires system instrumentation and A3E-Depth-First [26] app instrumentation, which are both incompatible with our code coverage measurement. Therefore, we compare `TextExerciser` with the rest three state-of-the-art tools that do not need any instrumentations. Note that both Sapienz and A3E-Depth-First adopt random text generation as shown in Table III—the results will be similar to the tested three exercisers.

Our settings of three existing tools are as follows. During our experiment, we configure Monkey with a fixed-event seed as documented by Continella et al. [34] so that Monkey will always explore the same sequence of events, e.g., clicking on the same position during different runs. Such a configuration will mitigate randomness that is introduced in comparing Monkey with `TextExerciser` and random text input generation. We configure DroidBot via manually writing text inputs based on the principle illustrated in their paper [22] for DroidBot. We adopt the default configuration of Stoat together with their own random text input generation.

1) Experiment Setup: We now describe our dataset, i.e., 40 Android apps in Table IV, used for comparing existing exercisers. Here is how we select these apps. We choose top 500 apps in terms of download amount from all the categories except for games from Google Play and form all these apps into a dataset. Then, we randomly select 1,200 apps for our hint analysis in Table I—the rest is further filtered to ensure that their required Android version is lower than 4.3 so that they can be used in the sensitive behavior detection in §V-B. Next, we apply Ella [35] to measure code coverage of all the rest apps; if Ella cannot instruments these apps, particularly Yippi and BlackWhiteMeet, we apply miniTracing [36] instead. Note that both Ella and minTracing, just like many other instrumentation tools, cannot cover and instrument native code.

We then introduce our experiment environment, i.e., four OnePlus 6T mobile phones with the same device configuration (Android 9.0, System build number A6010_41_181115). All devices are connected with servers, either a Windows server 2018 or a Ubuntu 16.04, via Android Debug Bridge (ADB) [37]. We choose the server platform based on the corresponding tools' requirement. Each experiment runs for one hour, a reasonable exercising time that is also adopted by prior work [19], [25], [33], and is repeated for three times

Table IV

OVERVIEW OF THE INSTRUMENTED TOP APPS USED FOR TESTING EFFECTIVENESS OF MONKEY, STOAT, DROIDBOT AND TEXTEXERCISER. THE NUMBER OF INSTRUMENTED FUNCTIONS ARE LISTED IN THE COLUMN "#METHOD", AND THE "#ACTIVITY" ARE CALCULATED BY EXTRACTING THE ACTIVITY NAMES FROM THE ANDROIDMANIFEST.XML OF EACH APP. WE ALSO MANUALLY VERIFY WHETHER AND HOW AN APP SUPPORT LOGIN. ○: THIS APP DOESN'T SUPPORT LOGIN, ◐: THIS APP NEED SIGN UP BEFORE LOGIN ◑: THIS APP ONLY CAN LOGIN WITH 3RD PARTY ACCOUNTS LIKE FACEBOOK ACCOUNT. ●: THIS APP CAN LOGIN WITH REGISTERED ACCOUNTS OR 3RD PARTY ACCOUNTS.

ID	App Name	Category	Down.	Login	#Activity	#Method	ID	App Name	Category	Down.	Login	#Activity	#Method
01	AndSMB	Productivity	1M+	○	17	17,410	21	Atomy	Business	1M+	◐	6	1,088
02	AutoScout24	Vehicles	10M+	◐	40	104,044	22	DealDash	Shopping	5M+	●	58	44,946
03	BeautifulHairstyle	Beauty	1M+	○	7	3,715	23	Cram	Education	1M+	●	66	17,390
04	CV-Library	Business	500K+	◐	22	24,603	24	Unfollowers	Social	1M+	◑	5	12,314
05	EasyMobileRecharge	Shopping	1M+	●	42	13,187	25	Meet24	Dating	5M+	●	62	26,272
06	Eskimi	Social	1M+	●	7	31,855	26	Schoolcalc	Education	1M+	○	4	947
07	FiltersorSelfie	Beauty	1M+	○	8	2,884	27	Callernamepro	Lifestyle	1M+	○	5	242
08	FloorPlanCreator	Art	5M+	○	13	8,848	28	MybabyPiano	Parenting	5M+	○	3	727
09	Hdr.Lite	Photography	1M+	○	23	13,807	29	Tapeatalk	Productivity	1M+	○	9	3,286
10	InNote	Tools	1M+	○	5	7,915	30	Hanjahandwritingrecog	Education	1M+	○	6	3,827
11	LINECamera	Photo	100M+	○	64	83,215	31	Writeonimage	Photography	1M+	○	7	843
12	OfficeSuite	Business	100M+	●	117	283,841	32	DocsToGo	Business	50M+	◐	31	32,148
13	Speedometer	Auto & Vehicles	1M+	○	11	17,031	33	Healthplus	Health & Fitness	1M+	●	76	22,145
14	Stock Watch	Finance	1M+	○	33	2,606	34	Coco	Communication	10M+	◐	126	65,677
15	Wakelockdetector	Productivity	1M+	○	8	669	35	SuperShuttle	Travel & Local	500K+	◐	57	26,125
16	HeartRateMonitor	Health	1M+	○	5	786	36	Saviry	Shopping	100K+	●	20	5,652
17	CurrencyConverter	Tools	1M+	○	2	253	37	10times	Events	100K+	●	88	33,002
18	StatusDownloader	Tools	1M+	○	8	2,849	38	Flipboard	News & Magazines	500M+	●	70	27,527
19	BestHairstyles	Beauty	5M+	○	4	5,704	39	Yippi	Social	500K+	◐	171	214,998
20	Yandex	Shopping	10M+	●	26	3,5480	40	BlackWhiteMeet	Dating	100K+	◐	221	113,299

to reduce randomness. After each experiment, we will revert the app's private data and modifications to shared resources, e.g., SD card and data stored in system services. Note that because ADB may be offline after a long time, our Xposed module, which is responsible for validation code, will also automatically reboot the Android system and reconnect the ADB if this happens.

In the following, we measure the code coverage of Android apps via two detailed metrics, i.e., method and activity coverage. Method coverage refers to the number of methods instrumented by Ella or miniTracing and triggered during exercising, and activity coverage the number of triggered activities on the stack during exercising, which are registered in the AndroidManifest.xml.

2) Code Coverage of Popular Android apps: Table V shows the results of method and activity coverage on 40 apps achieved by three existing exercisers. On average, the code coverage of `TextExerciser` is higher than or on par with the default text exerciser, i.e., either random or predefined. Particularly, `TextExerciser` triggers 48.5% more activities and 29.0% more methods for Monkey, 37.0% more activities and 20.2% more methods for DroidBot, and 45.3% more activities and 26.4% more methods for Stoat.

It is worth noting that the number of triggered methods by Stoat+`TextExerciser` is sometimes smaller than the one by Stoat+Random especially when the text inputs are with less constraints and can be easily fulfilled. The reason of such a slight decrease is that Stoat written in Ruby needs to communicate with `TextExerciser` written in Python via a slow, relatively-inefficient command line pipe. This unnecessary overhead will sometimes affect the code coverage under our strictly one-hour testing. Particularly, we find that such an overhead of Stoat+`TextExerciser` is 8.3 mins for each app on average, i.e., two times more than the one of Monkey+`TextExerciser` (3.1 mins) and DroidBot+`TextExerciser` (3.4 mins).

B. Improvement of Existing Dynamic Analysis

In this section, we answer RQ2 (*can `TextExerciser` improve existing dynamic analysis of Android apps?*) by evaluating `TextExerciser` on existing dynamic security analysis such as sensitive behavior detection and privacy leak detection. We use `TextExerciser` to drive existing dynamic analysis tools shown in Table XI and see whether we can observe any improvement when compared with prior Android exercisers. Particularly, we choose two open-source tools—namely TaintDroid [11], a taint analysis of tracking dataflows between sensitive APIs, and ReCon [23], a traffic analysis of detecting privacy information in network packets—out of many existing dynamic analysis tools [9], [11], [23], [38]. We also implemented a keyword-based traffic analysis that searches for the existence of keywords in network packets for privacy leaks—all the used keywords are shown in Table VI.

1) Experiment Setup: Our experiment is conducted on two Android Nexus 4 phones with Android 4.3 systems because of the requirement of TaintDroid and ReCon. Since DroidBot is incompatible with this runtime environment, we only compare `TextExerciser` with Monkey and Stoat. The apps are the same as these popular ones in §V-A and the exercising time limit is also one hour.

2) Results of Privacy Leak Detection: The results of privacy leak detection are shown in Figure 7. Note that we removed all the redundant results and only kept privacy leaks with unique source and sink for TaintDroid. A high-level observation is that all dynamic analysis detects more privacy leaks with the help of `TextExerciser`. We manually check some of these found new privacy leaks, which are due to two reasons. First, since `TextExerciser` can generate valid inputs to satisfy the input restrictions during the exploration of Android apps, `TextExerciser` can explore deep code branches of Android apps, and trigger more critical app behaviors. Second, because many privacy leaks occur when users input personal information to the app, `TextExerciser` can

Table V

THE METHOD AND ACTIVITY COVERAGE ON 40 INSTRUMENTED APPS OF THE THREE EXERCISERS, I.E., MONKEY (M.), STOAT (ST.) AND DROIDBOT (D.), WITH DIFFERENT TEXT INPUT GENERATION STRATEGY, I.E., RANDOM (R.), TEXTEXERCISER (TE) AND PREDEFINED (P.). WE ADOPT THE DEFAULT TEXT INPUT GENERATION STRATEGY OF THESE TOOLS WHEN COMPARING WITH TEXTEXERCISER. NOTE THAT ↑ AFTER TEXTEXERCISER NUMBER MEANS THAT THE IMPROVEMENT IS LARGER THAN 1%, - THE DIFFERENCE IS WITHIN 1%, AND ↓ THE DECREASE IS LARGER THAN 1%.

ID	#Triggered Activity						#Triggered Method					
	M.+R.	M.+TE	St.+R.	St.+TE	D.+P.	D.+TE	M.+R.	M.+TE	St.+R.	St.+TE	D.+P.	D.+TE
01	6.0±0.0	7.3±0.5 ↑	5.3±0.5	6.0±0.8 ↑	4.7±0.5	4.7±0.5 -	933±32	990±111 ↑	662±13	824±145 ↑	661±9	671±25 ↑
02	2.3±0.5	3.0±0.0 ↑	2.0±0.0	2.0±0.0 -	1.3±0.5	3.0±0.0 ↑	17,332±1,267	19,471±726 ↑	18,548±637	19,307±106 ↑	16,662±33	17,498±998 ↑
03	3.7±0.5	3.7±0.5 -	3.0±0.0	3.0±0.0 -	3.0±0.0	3.0±0.0 -	1,372±77	1,378±75 ↑	1,194±55	1,250±2 ↑	1,292±31	1,302±13 ↑
04	1.0±0.0	1.0±0.0 -	1.0±0.0	1.3±0.5 ↑	1.0±0.0	1.0±0.0 -	1,829±49	1,848±28 ↑	1,637±73	1,856±10 ↑	1,835±17	1,894±42 ↑
05	2.3±0.5	4.3±0.5 ↑	4.7±0.5	7.3±3.3 ↑	4.7±0.5	16.3±2.4 ↑	1,701±187	1,875±37 ↑	1,845±46	2,079±111 ↑	1,901±25	2,418±75 ↑
06	2.0±0.0	3.0±0.0 ↑	2.0±0.0	2.3 ±0.5 ↑	2.0±0.0	3.3±0.5 ↑	3,632±663	6,455±47 ↑	4,212±26	5,795±1,423 ↑	4,365±42	6,340±232 ↑
07	3.0±0.0	3.0±0.0 -	1.0±0.0	2.0±0.0 ↑	3.0±0.0	3.0±0.0 -	1,021±13	1,106±48 ↑	209±46	883±21 ↑	1,076±46	1,146±10 ↑
08	6.3±0.5	6.3±0.5 -	5.0±0.8	5.3±1.2 ↑	4.0±0.0	4.0±0.0 -	3,279±77	3,305±46 -	2,337±289	2,700±275 ↑	2,704±11	2,710±17 ↑
09	5.3±0.9	5.3±0.9 -	7.0±0.8	8.3±0.9 ↑	6.0±0.0	6.0±0.0 -	900±82	971±130 ↑	1,672±162	1,766±93 ↑	872±202	1,063±1 ↑
10	2.0±0.0	2.3±0.5 ↑	3.0±0.0	3.0±0.0 -	2.0±0.0	2.0±0.0 -	1,713±18	1,768±68 ↑	1,695±33	1,624±8 ↓	1,482±1	1,482±1 -
11	9.7±0.5	9.7±0.5 -	9.3±1.2	11.0±1.6 ↑	5.3±0.5	6.0±1.4 ↑	15,314±656	16,404±1,180 ↑	19,021±335	19,687±636 ↑	10,125±90	10,254±293 ↑
12	8.0±0.8	10.7±0.5 ↑	9.0±3.3	13.3±2.1 ↑	11.0±1.4	12.0±2.2 ↑	47,828±4,155	53,414±1,126 ↑	30,667±6,294	41,584±7,130 ↑	46,256±1,552	53,183±5,846 ↑
13	5.0±0.8	5.3±0.5 ↑	5.3±0.9	5.3±0.5 -	2.0±0.0	2.0±0.0 -	4,375±213	4,491±116 ↑	5,291±151	5,172±62 ↓	4,378±26	4,661±261 ↑
14	11.0±0.0	11.0±0.0 -	7.7±1.7	8.7±0.5 ↑	11.0±0.8	11.0±0.0 -	839±87	963±36 ↑	784±135	833±20 ↑	982±254	1,233±48 ↑
15	4.3±0.5	4.3±0.5 -	3.0±1.4	4.3±0.5 ↑	2.0±0.0	2.7±0.5 ↑	208±27	224±12 ↑	216±47	249±22 ↑	180±1	216±22 ↑
16	2.0±0.0	2.0±0.0 -	3.0±0.0	3.0±0.0 -	3.0±0.0	3.0±0.0 -	169±2	175±0 ↑	326±5	324±12 -	277±0	277±0 -
17	1.3±0.5	1.3±0.5 -	1.0±0.0	1.0±0.0 -	1.0±0.0	1.3±0.5 ↑	120±0	120±0 -	116±2	119±0 ↑	107±6	111±2 ↑
18	4.3±0.9	5.0±0.0 ↑	5.0±0.0	5.0±0.0 -	5.0±0.0	5.0±0.0 -	843±5	847±1 -	882±1	855±9 ↓	864±7	865±6 -
19	3.0±0.0	3.0±0.0 -	3.0±0.0	3.0±0.0 -	3.0±0.0	3.3±0.5 ↑	2,304±2	2,304±0 -	2,282±9	2,280±11 -	2,254±11	2,268±5 ↑
20	4.7±0.9	5.3±0.5 ↑	5.0±0.8	8.0±0.8 ↑	4.7±0.5	7.7±0.5 ↑	12,391±624	13,325±539 ↑	13,266±377	13,467±182 ↑	10,870±208	14,315±247 ↑
21	2.0±0.0	2.0±0.0 -	2.0±0.0	2.0±0.0 -	2.0±0.0	2.0±0.0 -	290±8	319±49 ↑	244±16	260±6 ↑	278 ±9	283±7 ↑
22	6.3±0.5	17.0±2.9 ↑	6.3±3.1	8.0±0.0 ↑	9.0±0.0	11.0±1.4 ↑	6,609±64	10,715±690 ↑	6,597±387	6,785±41 ↑	6,822±29	8,677±383 ↑
23	6.7±0.5	11.0±0.8 ↑	10.3±1.2	14.3±1.9 ↑	11.0±1.4	12.7±1.9 ↑	1,539±440	2,324±114 ↑	2,103±325	2,796±335 ↑	1,488±511	1,986±592 ↑
24	2.0±0.0	2.0±0.0 -	2.0±0.0	2.0±0.0 -	2.0±0.0	2.0±0.0 -	1,988±214	2,278±0 ↑	2,354±40	2,440±162 ↑	2,257±1	2,268±14 -
25	6.0±0.0	18.0±2.2 ↑	6.3±0.5	17.3±0.9 ↑	7.0±0.0	21.7±2.1 ↑	3,264±269	5,441±115 ↑	3,803±9	5,512±16 ↑	3,845±48	6,207±403 ↑
26	1.0±0.0	1.0±0.0 -	3.0±0.0	3.0±0.0 -	3.0±0.0	3.0±0.0 -	407±1	407±1 -	404±0	404±0 -	390±10	406±6 ↑
27	4.0±0.0	4.0±0.0 -	4.3±0.5	4.3±0.5 -	4.0±0.0	4.0±0.0 -	115±2	115±2 -	118±1	112±0 ↓	121±1	123±1 -
28	1.0±0.0	1.0±0.0 -	1.0±0.0	1.0±0.0 -	1.0±0.0	1.0±0.0 -	336±10	348±0 ↑	301±17	324±4 ↑	344±7	346±10 -
29	3.7±0.5	4.0±0.0 ↑	2.7±0.9	3.7±0.5 ↑	2.0±0.0	2.0±0.0 -	415±17	414±8 -	274±110	773±18 ↑	338±3	336±0 -
30	3.0±0.0	3.0±0.0 -	3.0±0.0	3.0±0.0 -	4.0±0.0	4.0±0.0 -	788±9	788±17 -	387±12	385±11 -	423±20	451±2 ↑
31	4.0±0.0	4.0±0.0 -	5.0±0.0	5.0±0.0 -	5.0±0.0	5.0±0.0 -	79±3	87±14 ↑	115±0	111±2 ↓	149±3	151±0 -
32	4.3±0.5	5.3±0.5 ↑	5.0±0.8	6.7±0.9 ↑	2.3±0.5	3.0±0.0 ↑	3,320±114	6,613±792 ↑	5,894±109	5,884±68 -	3,386±623	4,616±26 ↑
33	4.0±0.0	9.3±1.9 ↑	5.0±0.0	8.0±1.6 ↑	4.0±0.0	6.0±0.0 ↑	1,807±127	3,516±320 ↑	1,904±100	2,805±156 ↑	1,956±11	1,963±7 -
34	3.3±0.5	13.0±1.6 ↑	8.0±2.9	10.3±1.7 ↑	5.0±0.0	9.3±4.8 ↑	3,341±220	7,946±299 ↑	4,738±2,043	8,046±656 ↑	3,375±20	5,237±2,361 ↑
35	4.0±0.0	4.3±0.5 ↑	4.0±0.0	4.7±0.5 ↑	4.0±0.0	15.3±0.9 ↑	4,016±212	5,123±36 ↑	4,321±44	5,221±13 ↑	4,303±1	7,428±50 ↑
36	6.3±0.9	7.7±0.5 ↑	6.3±0.9	6.3±0.5 -	12.0±0.0	14.3±0.9 ↑	1,684±142	1,784±27 ↑	1,773±39	1,818±50 ↑	1,944±17	2,045±51 ↑
37	5.0±0.8	16.0±0.0 ↑	7.0±0.0	9.7±2.1 ↑	7.0±0.0	13.3±2.1 ↑	4,171±151	7,779±44 ↑	4,378±45	6,648±1,482 ↑	4,322±22	7,885±286 ↑
38	5.0±0.8	7.0±0.0 ↑	4.7±0.5	10.3±0.5 ↑	15.0±0.8	15.3±0.9 ↑	4,667±223	8,021±86 ↑	4,485±609	4,823±159 ↑	9,446±260	9,876±177 ↑
39	2.0±0.0	12.7±0.9 ↑	2.0±0.0	12.5±1.5 ↑	2.0±0.0	16±0.8 ↑	6,456±303	12,705±850 ↑	5,488±279	10,297±537 ↑	6,215±237	13,721±412 ↑
40	4.7±0.5	7.7±0.9 ↑	4.0±0.0	7.0±1.4 ↑	5.3±0.5	9.3±0.5 ↑	5,800±144	10,083±690 ↑	2,869±72	3,435±726 ↑	4,616±823	10,869±363 ↑
Total	166.2±12.8	246.8±18.8	177.0±23.2	242.5±27.6	186.3±7.8	270.7±25.1	169,223±10,908	218,239±8,478	159,410±12,995	191,534±14,716	165,162±5,227	208,780±13,296
	-	+48.5%	-	+37.0%	-	+45.3%	-	+29.0%	-	+20.2%	-	+26.4%

Table VI

THE CATEGORY OF PRIVATE INFORMATION IN TAINTDROID, RECON AND A KEYWORD-BASED TRAFFIC ANALYSIS IMPLEMENTED BY OURSELVES. WE LIST ALL THE USED KEYWORDS IN THE LAST COLUMN.

Category	TaintDroid	ReCon	Keyword-based traffic analysis
Device Identifier	Accelerometer IMEI, IMSI ICCID Serial Number	MAC Address Device ID Advertiser ID IOS IFA ID	MAC Address, IMEI, IMSI, ICCID, Device Serial Number
User Identifier	Phone Number Microphone Camera Browser History	Name Gender Birthday E-mail Relatioship Status	Birthday, Date Of Birth, Expire, Expiration Year/Month/Day, Year, Month, Day, Date, Birth, Username, First Name, Last Name, User ID, Nick Name, Skype Name, Name, Nick, Email, Mailbox, Email Address, Mail
Contact Information	Contact Provider	Phone Number Address Book	Contact/Cell/Phone/Mobile/Full number, Phone No, Mobile Phone, Country Code, CC
Location	GPS Location Net-base Location Last-known Location	Zip Code	Postal/Post Code, ZipCode, ZipNumber, Location, Country, City, Address, Area, Region
Credentials	Account Information	Username Password	Id, Account Number, Partner Code, Password, Passwd, PassCode Verify, Verification, Check Number, Verification/Verify Code, Verification Number, Otp, PinView, Pin, Pinnum, Account Pin, Pin Code, PinNumber, Pin Number, Card Number, Account Name,

successfully generate such inputs and trigger the privacy leak.

Next, we compare the detection capability of these three tools and start from TaintDroid and the other two. TaintDroid detects the most number of privacy leaks, because TaintDroid, comparing with ReCon and keyword-based search, can detect encrypted private identifiers. On the contrary, ReCon and keyword-based search detect more categories of private information, especially user credentials, because it is hard to mark the sources of these information in taint analysis. A detailed breakdown of these detected privacy leak categories can be found in Appendix C.

We then compare ReCon and keyword-based search. Recon detects one more user identifier than the keyword-based search because some keywords, such as "user", are very common and cannot be used for detection—that is the limitation of a keyword-based search. When using Stoat, the keyword-based search detects seven more privacy leaks due to special encoding like "@" as "%40". It is worth noting that the false positive rate of ReCon is relatively high mainly due to the fact that ReCon often mistakenly considers a large port number as a zipcode during our manual inspection.

3) Case Study: In this part, we present a case study of the apps with additional privacy leaks found by TaintDroid and ReCon with the help of `TextExerciser`. Note that interestingly, as a byproduct, `TextExerciser` also finds some

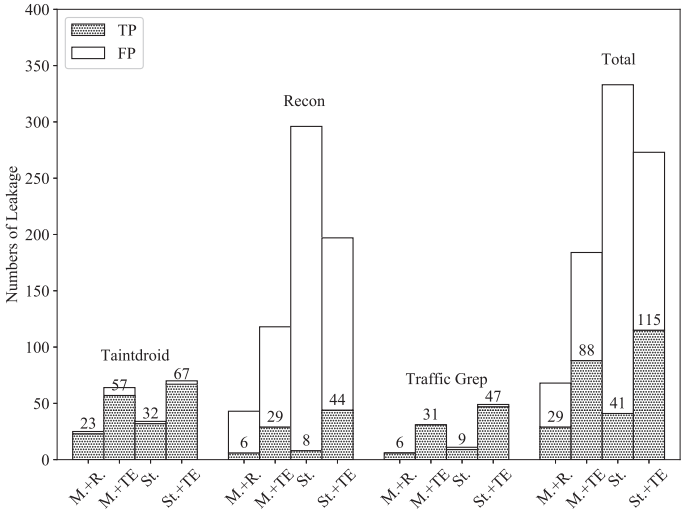


Figure 7. The number of detected privacy leakage of three dynamic analysis, i.e., TaintDroid, ReCon, and a keyword-based traffic search.

Table VII
ALL VULNERABILITIES AND BUGS DETECTED IN THE GOOGLE PLAY APPS BY USING TEXTEXERCISER AS EXERCISERS.

App Name	#Downloads	Description
Previously-unknown Vulnerabilities:		
BlackWhiteMeet	100,000+	Doesn't verify signature in https
Coco	10,000,000+	Leak user credential in http
10times	100,000+	Leak user location and device info in http
Yippi	100,000+	Change user password in http
Saviry	100,000+	Modify user profile in http
Eskimi	1,000,000+	Leak user credential and profile in http
Previously-unknown Bugs:		
Flipboard	500,000,000+	Unable to login after mobile sign-up
SuperShuttle	500,000+	Bypass constraints for phone number

bugs, e.g., malformed text constraints, of existing Android apps when generating text inputs. A list of these apps is shown in Table VII—We have responsibly disclosed all these issues to corresponding app developers.

a) Insecure Transfer of User Credential or Private Information: We describe three apps that transfer user credentials such as username and password in plaintext, i.e., via HTTP protocol.

- Coco, a popular social app with more than 10,000,000 downloads on Google Play. The 7.6.8 version of Coco, i.e., the latest at the paper's submission, has a vulnerability that transfers user credentials in plaintext without any encryption. This vulnerability involving two consecutive HTTP requests is triggered with the help of `TextExerciser` during the sign up and login pages of Coco. It is worth noting that all other network communications in Coco except these two adopt HTTPS protocol. We have reported the vulnerability to Coco's developer and a patched version is coming soon.
- Saviry, a shopping app with more than 100,000 downloads on Google Play and a 4.4 rating score. The latest version of Saviry has a vulnerability that transfers an HTTP post request in plaintext. Specifically, an active network attacker can directly modify the request to alter user's profile and a passive network attacker can also sniff the cookie and send another request to tamper user's profile. `TextExerciser` helps to discover this vulnerability by correctly filling all the fields in the user profile and then triggering the request.
- Eskimi, a social app to find and communicate with new friends. Eskimi transfer private information, such as birthday, gender, email, and city, as well as user credentials all in plaintext. `TextExerciser` helps to discover more private information transfers, such as birthday and email, on certain pages.

b) Insecure Configuration (i.e., Missing Certificate and Hostname Check) of SSL Communication: We describe BlackWhiteMeet, a dating app with more than 100,000 downloads on Google Play, which enables users to find new friends and communicate with each other. BlackWhiteMeet configures SSL communication incorrectly by missing certificate and hostname check so that an adversary can mimic the authentic server using a fake certificate.

c) Constraint Bugs for Checking Text Inputs: We describe two bugs of existing Android apps when checking text constraints. This is a byproduct of `TextExerciser` when it tries to generate text inputs—`TextExerciser` compares the generated text inputs with a common knowledge of that field, e.g., a phone number and email address. If the generated input violates the common constraints enforced by all other apps, we will consider it as a bug. Now, we introduce two concrete bug examples.

- Phone number validity checking bug in SuperShuttle, a travel app on Google Play. `TextExerciser` finds a bug that can bypass the original constraints enforced for phone numbers. More specifically, SuperShuttle requires user to enter a valid phone number along with country code, and the app server will check if the number and the country code match with each other. Once the check failed, SuperShuttle will notify the user to modify the input. However, `TextExerciser` finds that if we enter a much larger country code, for example "100000", any number entered as the phone number is considered as valid. This bug can be potentially exploited to register a large amount of fake or bot accounts with the app server.
- Sign-in bug in Flipboard, a news app on Google Play. The app has a bug in its mobile sign-in page: Specifically, if one signs up an account with the app server in its mobile app, he or she cannot log into the app using the newly registered account. One can only log into the app using an account registered in the web browser or keeps logged in using the newly registered account after sign-up in mobile devices. We confirm the bug via manually registering an account and trying to log into the app.

C. Tool Performance on Popular Android Apps

In this section, we answer RQ3 *(is TextExerciser efficient for generating text input for popular Android apps?)* by evaluating TextExerciser from four perspectives:

- *Coverage of syntax rules.* We evaluate the coverage of our rules in interpreting the syntax trees of all the hints from a larger dataset of randomly collected Android apps.
- *Code coverage of a larger number of Android apps.* We evaluate the capability of Monkey+TextExerciser and DroidBot+TextExerciser in analyzing these randomly collected Android apps and their corresponding code coverage.
- *Efficiency of input generation.* We evaluate the number of trials before TextExerciser can generate a valid input for a given text field.
- *Performance of learning model.* We evaluate the performance of the learning model used by TextExerciser with different parameters.

1) Dataset and Experiment Setup: In this part, we use a dataset of 6,000 randomly-collected, popular Android apps from Google Play. We conduct the experiment on 16 official Android x86 emulators with 4 CPU cores, 2 GB RAM and 2 GB SD card and the aforementioned four physical phones (OnePlus 6T), all of which run Android 9.0. We first run an app on Android emulator together with Monkey+TextExerciser or DroidBot+TextExerciser and if the execution fails, we will run the app in physical phones. We restrict each execution to be within 30 minutes. Note that for similar reasons stated in §V-B, we only evaluate Monkey/DroidBot+TextExerciser in this section.

Monkey/DroidBot+TextExerciser in total analyze 5,640, i.e., 94.0% of all the Android apps, which is the same as the results if we run the Monkey/DroidBot alone. In other words, TextExerciser does not bring any additional failed apps to the Monkey or DroidBot. The details are that 5,060 apps run correctly without crashing in Android emulators, and then additional 580 runs correctly in physical phones. We manually look at these fail cases—the major reason is that *aapt* does not extract the complete launcher information and thus the Monkey/Droidbot cannot automatically start the apps via Android ADB.

2) Syntax Rule Coverage: TextExerciser achieves 87.3% coverage when using the syntax rules to parse the hint texts in §III-C. Here are the details. TextExerciser extracts 328,282 text sentences from all the apps in the dataset. After phase 1 with a learning model, TextExerciser finds that 3,450 of them are input hints and the rest are general descriptive sentences of user interface. Then, TextExerciser successfully analyzes 3,012 of input hints, leading to a 87.3% coverage. After that, TextExerciser generates 3,993 constraints and 5.7 lines of Z3StrSolver code on average for each constraint.

Let us look at a successful example of utilizing syntax rules in Table II to parse input hints. TextExerciser finds an input hint ("Please choose a username that is at least 3 characters to sign up and doesn't contain special characters") in the app, rateME. Then, TextExerciser splits this long hint into two short hints with the conjunction word "and". Next, the first hint will be classified into *C1* ("Length Constraints"), and the second hint *C6* ("Input should not contain certain characters"). TextExerciser parses *C1* through the rule *LowerBound* and parses *C6* through *Exclusive* in Table II. Then, the parsed results will be passed to our input generation engine, and be combined together to solve an input.

Next, we look at some of the failure cases in parsing the hint text. First, some apps utilize unusual words or vague words to illustrate hints. For example, "Authentication failed". Second, some hint texts are embedded in popup figures. Since TextExerciser does not apply an OCR [39], it cannot identify texts in figures. Third, the popup window of some apps disappear so quickly that TextExerciser does not have enough time to obtain the texts.

Lastly, we show the category distribution for all the successfully analyzed 3,012 hints in Figure 9. The distribution is similar to the one of our training set of Android apps. *C14* ("Non-directional constraints") is the most popular category, because most apps tend to provide such a constraint first together with other directional ones to warn the users. *C1* ("The lower bound of input length") and *C15* ("Equivalence Constraints") come next, because passwords are widely used in many Android apps during the login phase.

3) Code Coverage of This Large Dataset: We evaluate the code coverage on this larger dataset in Figure 10. Let us start from the comparison with Monkey. On average, Monkey+TextExerciser triggers 24.0% activities in Android apps as opposed to 19.0% activities triggered by the Monkey with the default random text input generation. Furthermore, the upper bound of triggered activities of Monkey+TextExerciser is 80%, larger than 67%, i.e., the one of Monkey with a random strategy. We also compare TextExerciser with predefined strategy in DroidBot. DroidBot+TextExerciser triggers 25% activities as opposed to 20% by DroidBot+Predefined on average. The upper bounds of triggered activity of DroidBot+TextExerciser and DroidBot+Predefined are both 100%, but the absolute value of the upper bound in DroidBot+TextExerciser is 72 as opposed to 52 by DroidBot+Predefined.

We further break the code coverage based on the constraint categories by randomly selecting 10 apps with a certain constraint category and exercising these apps using DroidBot+TextExerciser and DroidBot+Predefined. Figure 8 shows the median (middle line), 25%–75% (bar), and top/bottom of activity coverage broken down by 18 constraint categories. TextExerciser is on par with or outperform predefined strategies in all the metrics across 18 constraint categories. It is worth noting that all the apps have many hints and corresponding categories and our evaluation ensures that at least the target hint category is involved in the selected 10 apps.

4) Number of Trials in Generating Valid Text Inputs: TextExerciser achieves about 95.1% success rate when

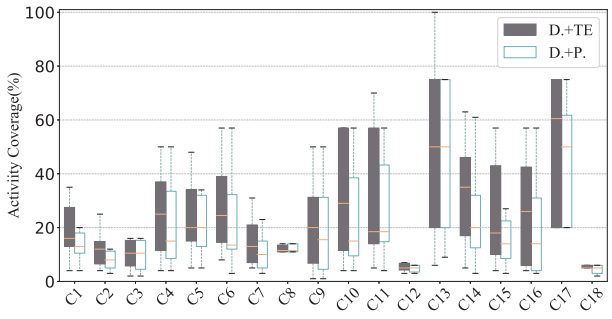


Figure 8. The activity coverage of `TextExerciser` and DroidBot on 10 apps in each hint category.

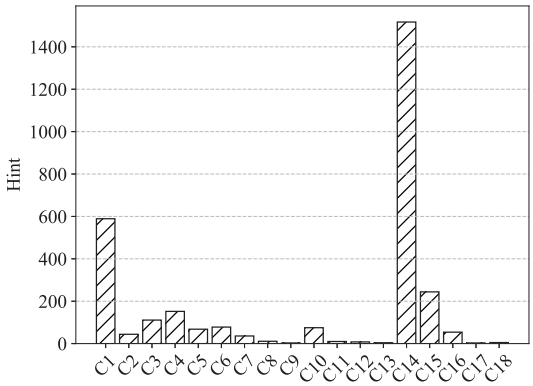


Figure 9. The category distribution of successfully-interpreted hints by `TextExerciser`. These categories, i.e., C1–18, are explained in Table I.

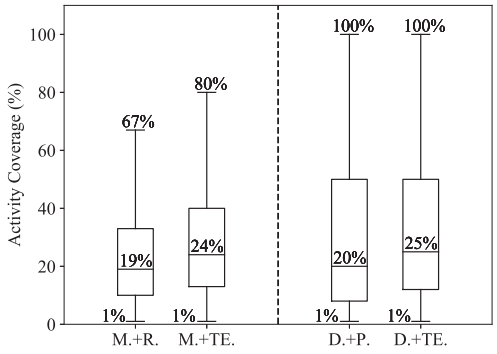


Figure 10. Activity coverage of Monkey (random and `TextExerciser`) and DroidBot (predefined and `TextExerciser`).

generating an input for a text field for the first time. Then, in the second trial, `TextExerciser` takes more feedbacks and achieves 96.7% success rate. In practice, most of the input generations are finished in three rounds. The number of trials is limited to 30 and only 1.2% of mutation will exceed this limitation.

Let us first see a concrete successful example, i.e., "Achievement - Rewards for Health", which needs three trials in generating a password. The hints for password generation of this app show up step by step. Specifically, in the first trial, `TextExerciser` does not identify any input hint and generate a random input "PA". After entering this input, the app prompts the first hint—"Minimum of 8 characters". Then, in the second trial, `TextExerciser` pares this hint and generates an input "HPPABEQH" using the solver. This input is still invalid and the app prompts another input hint—"Must contain at least 1 lowercase, 1 uppercase, and 1 number". Therefore, in the third trial, `TextExerciser` adds three more constraints to the solver code and generates an input "1PA07aAO", which finally satisfies the requirements of this input field.

Table VIII
BREAKDOWN OF REASONS THAT CAUSE FAILED INPUT GENERATION AFTER 30 TRIALS.

Type	#Hints	Reason for exceeding 30 trials
1	1.088%	Semantics-specific hints
2	0.80‰	Hints with loose constraints
3	0.32‰	Misclassified hints
Total	1.2%	

We then break down the failed cases, i.e., 1.2% of all the hints, based on their reasons and show them in Table VIII. There are three major reasons: (i) semantics-specific hints, (ii) hints with loose constraints, and (iii) misclassified hints. First, some hints are semantics specific, which require certain external knowledge to solve. Now, we illustrate an example, called "DQ Texas". The app's sign-up page requires username, password, and phone number, which can be exercised by `TextExerciser`, but more importantly also contains a so-called "invite code" field. The hint provided by the app for a random code is that "Please check your invite code and try again." A real user can either obtain the code from DQ store or search one online; by contrast, `TextExerciser` cannot generate one without knowing the semantic meaning.

Second, there are some hints that express a constraint that is looser than what has been actually enforced. We believe that these are unfriendly designs of user interfaces. Here is one example, called "TeamLease". Once someone inputs an invalid phone number, the app shows a hint saying that "please enter valid 10 digital". However, the actual constraint enforced by the app for the phone number field is 10 digits plus a country code with "+91". `TextExerciser` cannot solve this constraint without a proper hint.

Lastly, because `TextExerciser` adopts a machine learning model to classify hints, misclassifications are inevitable, especially for these hints with formats that are not seen in the training dataset. These misclassifications will also lead to failed generation of inputs. Note that `TextExerciser` may still be able to generate inputs if one hint is misclassified because other hints may still help `TextExerciser` in the generation.

5) Performance of Hint Classifier: In this section, we evaluate the performance of our hint classifier, an open-source multi-class CNN-RNN model [27], trained from dataset mentioned in §IV. We tried to adjust all the parameters, such as batch size, the number of hidden units, and max pooling, but observed that the default parameters provided by the tool are still the best ones in our problem. Therefore, we adopted their default setting as shown in Appendix B. Our evaluation

results of this model against testing set show that the accuracy, precision and recall are 90.2%, 89.4% and 90.2% respectively.

It is worth noting that false positives have less impact on the performance of TextExerciser because even if a non-hint is misclassified as a hint, TextExerciser will just add more constraints but still generate valid inputs. False negatives have a relatively higher impact but 90.2% recall is sufficient because even if one hint is misclassified, other hints may still help TextExerciser to generate a valid input.

VI. Discussion

We discuss some practical issues in implementing and deploying TextExerciser in real world.

Supported Language. Our prototype of TextExerciser supports only English apps, but can be extended to apps in other languages. As a proof of concept, we extend TextExerciser to 10 non-English apps using Google Translate to convert the hints to English. Our evaluation shows that TextExerciser solves 20 of 21 hints extracted from these apps and then successfully generates inputs based on these hints. Details of these apps can be found in Appendix A.

Apps with WebView. TextExerciser relies on UiAutomator [30] to identify and capture WebView widgets and corresponding embedded hints. We manually checked 150 popular apps excluding games, i.e., those mentioned in the introduction, and the results show that 25 apps contain WebView widgets and 23 of these widgets are correctly captured by UiAutomator and TextExerciser in exercising.

Server- vs. client-side validation. We perform a small-scale, manual experiment to compare the number of server- vs. client-side input validations. Our methodology is as follows. We first use a mobile app with network connection enabled and record the appeared hints, and then repeat the same procedure with the network connections disabled. We perform the experiment on the 150 popular apps excluding games as mentioned in the introduction and the results show that 86 out of 649 hints in these apps correctly displayed without network connection, i.e., implemented purely at the client-side, and the rest, which is 563 hints, requires more or less server support.

Result randomness. We reduce the randomness in our experiment results via two methods: (i) fixing random seed, and (ii) repeating experiments. Specifically, we fix the random seed of Monkey during different runs in our experiment and also repeat every experiment for three times. It is worth noting that randomness cannot be fully mitigated due to many other factors, such as network delay. For example, even if we fix the seed of Monkey, network delay may cause a login page shows up after the next event is triggered. Therefore, all the follow-up events may be influenced as well.

Exclusion of Gaming Apps from Our Evaluation Dataset. We exclude gaming apps from our evaluation due to two reasons. First, gaming apps commonly utilize figures to illustrate notifications and most of their contents are presented in the form of images. Since this paper focuses on the text input generation, image-based game apps are apparently beyond the scope. We will release the collected apps for the convenience of other studies on text input generation of Android apps. Second, gaming apps usually heavily rely on native code for performance reasons—it is hard to measure the code coverage in native code.

Hint Obfuscation. TextExerciser requires that Android apps provide enough and clear hints for text inputs—this is reasonable because these hints are intended to provide to users so that they can interact with the app. We do find that some apps provide hints via figures and voices, which cannot be recognized by TextExerciser. Currently, the evaluation result shows that only about 1.4% of hints are missed by TextExerciser. In the future, we plan to introduce OCR [39] and voice recognition [40], and further understand these hints in the image or audio format.

VII. Related Work

In this section, we review related prior researches.

A. Input Generation in Testing Android Apps

Traditionally, a plenty of work focus on generating testing input for Android apps, especially in automating dynamic analysis. However, they either focus on event based input such as UI event and system event, or rely on a plenty of manual effort to generate valid text input. For example, Monkey [14], the most frequently used Android testing tool, only can generate UI events like randomly clicking elements on UI screen. Dynodroid [16] and Mulliner et al. [41] expands the UI events with system events, such as SMS receiving. However, when encountering a text input field like password, they must pause the testing and wait for manual input.

Some modern works, such as Sapienz [25], A3E-Depth-First [26], DroidBot [22], AppsPlayground [18] and Droid-DEV [19], fulfill text input fields by searching in a set of pre-defined input. If none of the pre-defined inputs can satisfy an input's constraints, these prior works will fail to exercise beyond this input. Another thread of work, i.e., Liu et al. [20], utilize RNN to train a learning model and use it to generate text input based on the app context. Unfortunately, it requires a large amount of manual effort to write input for training such a model. As a comparison, TextExerciser first identifies the input restrictions from UI screen by combining machine learning with UI structural analysis and then generates a text input with a mutation based strategy. The main advantage is that TextExerciser iteratively generate inputs for a given text field—i.e., even if a particular input fails, TextExerciser can still generate more inputs based on newly-collected hints as feedback.

In addition, another work Mobolic [42] uses symbolic execution to extract the input constraints in app code and utilizes a solver to generate valid input. However, many input checks are enforced at the server side of Android apps.

As a comparison, TextExerciser can generate inputs even if these input checks are performed at server side, because input hints are eventually shown at client-side app.

B. Widget Identification in Android Apps

UI widget identification is often combined and used together with Android app testing because an exerciser needs to interact with different Android UI widgets. SUPOR [43] and UIPicker [44] extract UI widget information from the layout's XML file and then identify sensitive inputs. UiRef [45] improves prior works by adopting a hybrid approach that combines both static and dynamic identifications: the static method identifies widgets from layouts, just like prior work and the dynamic method extracts each rendered layout during on-device execution. Similarly, CuriousDroid [46] instruments the Dalvik virtual machine to obtain the UI widgets and generate UI-related events during execution.

All the prior works on UI widget identification can be combined with `TextExerciser` in testing Android apps as widget identification is an orthogonal problem. `TextExerciser` adopts UIAutomator because of two major reasons. First, UIAutomator obtains all the widgets information dynamically during execution, which has incorporated many advantages claimed by prior works. Second, UIAutomator is open-source and compatible with many real-world Android apps.

C. Text Input Generation in Web

Input generation is a critical problem in testing web applications and locating vulnerabilities such as SQL Injection and XSS vulnerabilities. Based on their requirements for the source code of web apps, we can classify the prior work into two categories. The first part of work utilize white box testing to launch analysis on targeted web apps. For example, ACTEve [47] and S3 [48] first use symbolic execution to extract input constraints in the source code and then use a solver to generate an input. As a comparison,`TextExerciser` works for Android apps and does not require any source code—some of the source code is at the server side and unavailable to `TextExerciser`.

Another thread of related work [49], [50] leverage black box testing, but use manual effort to pre-define text inputs. For example, one of many vulnerabilities studied Vieira et al. [50] is to exploit web services using Acunetix web vulnerability scanner [51] with pre-defined username and password combinations.

These works are only available for generating some particular text fields, such as password, which has a public database. In many cases like salary, username, and ages, such a public database is unavailable. As a comparison, `TextExerciser` works on Android apps and relies on hints as a feedback to generate all the text inputs.

D. Fuzzing based Approach in Android Dynamic Analysis

Fuzzing is widely used in Android dynamic analysis. Commonly, the state-of-art approaches generate their input based on a bunch of domain knowledge about the input structures. For example, prior works [52]–[55] focus on fuzzing critical data structures in Android such as Intent and Binder, which are well-documented. In addition, another thread of work like Caiipa [56] use synthesized context observed in the wild to guide its fuzzer so that it can cover different context variations. As a comparison, `TextExerciser` focuses on exercising text inputs, which are not target of existing Android fuzzers. We can consider `TextExerciser` as a fuzzer on text inputs, but the text-based fuzzer is guided by feedbacks, i.e., these hints, provided by Android apps.

VIII. Conclusion

In this paper, we propose `TextExerciser`, an iterative, feedback-driven text input exerciser, which generates text inputs for Android apps. `TextExerciser` relies on a key insight that Android apps often provide feedback, called hints, for malformed inputs from users—at the same time, `TextExerciser` can also utilize such hints to improve the input generation.

Our evaluation shows that `TextExerciser` can achieve significantly higher code coverage than these tools with default text input generators. We also combine `TextExerciser` with existing dynamic analysis tools like TaintDroid and ReCon and show existing dynamic analysis tools are able to detect more malicious behaviors with `TextExerciser` than with existing exercisers. `TextExerciser` together with existing dynamic analysis tools is able to find several previously-unknown vulnerabilities in popular Android apps, such as user credential leakage in a social app, arbitrary user profile modification in a shopping app, and a software bug in another traveling app.

IX. Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1736208, U1836210, U1836213, 61972099, 61602121, 61602123), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," NDSS, 2018.

[2] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in CCS. ACM, 2017.

[3] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in NDSS, 2014.

[4] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in SPSM. ACM, 2012.

[5] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in USENIX Security, 2017.

[6] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in NDSS, 2015.

[7] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in CCS. ACM, 2013.

[8] M. Zheng, M. Sun, and J. C. Lui, "Droidtrace: A ptrace based android dynamic analysis system with forward execution capability," in IWCMC. IEEE, 2014.

[9] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in CCS. ACM, 2016.

[10] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in USENIX Security, 2012.

[11] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," TOCS, 2014.

[12] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in USENIX Security, 2014.

[13] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements,... and privacy leaks," 2018.

[14] Google. (2019) Ui/application exerciser monkey. Https://developer.android.com/studio/test/monkey.

[15] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware." in NDSS, 2016.

[16] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in FSE. ACM, 2013.

[17] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in EuroSys. ACM, 2014.

[18] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in CODASPY. ACM, 2013.

[19] Y. L. Arnatovich, M. N. Ngo, T. H. B. Kuan, and C. Soh, "Achieving high code coverage in android ui testing via automated widget exercising," in APSEC. IEEE, 2016.

[20] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in ICSE. IEEE, 2017.

[21] 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," in FSE. ACM, 2017.

[22] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in ICSE-C. IEEE, 2017.

[23] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "Recon: Revealing and controlling pii leaks in mobile network traffic," in MobiSys. ACM, 2016.

[24] Textexerciser open source address. Https://github.com/yyyyHe/TextExerciser.

[25] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in ISSTA. ACM, 2016.

[26] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in SIGPLAN Notices. ACM, 2013.

[27] jiegzhan. (2018) Multi class text classification cnn rnn. Https://github.com/jiegzhan/multi-class-text-classification-cnn-rnn.

[28] T. S. N. Group. (2019) The stanford natural language processing group. Https://nlp.stanford.edu.

[29] M. Research. (2017) Z3str3 string constraint solver. Https://sites.google.com/site/z3strsolver/.

[30] Google. (2019) Android ui automator. https://developer.android.com/training/testing/ui-automator.

[31] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," JAIR, 2002.

[32] rovo89. (2019) Xposed framework. Https://www.xda-developers.com/xposed-framework-hub/.

[33] 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," in ASE. ACM, 2018.

[34] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis." in NDSS, 2017.

[35] saswatanand. (2016) Binary instrumentation of android apps. Https://github.com/saswatanand/ella.

[36] T. Gu. Ape: Automated testing of android applications with abstraction refinement. http://gutianxiao.com/ape/#minitracing.

[37] Google. (2019) Android debug bridge. Https://developer.android.com/studio/command-line/adb.

[38] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," in Euro S&P. IEEE, 2017.

[39] Wikipedia. (2019) Optical character recognition. https://en.wikipedia.org/wiki/Optical_character_recognition.

[40] ——. (2019) Speech recognition. https://en.wikipedia.org/wiki/Speech_recognition.

[41] C. Mulliner and C. Miller, "Fuzzing the phone in your phone," Black Hat USA, 2009.

[42] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "Mobolic: An automated approach to exercising mobile application guis using symbiosis of online testing technique and customated input generation," SPE, 2018.

[43] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "{SUPOR}: Precise and scalable sensitive user input detection for android apps," in USENIX Security, 2015.

[44] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in USENIX Security, 2015.

[45] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie, "Uiref: analysis of sensitive user inputs in android applications," in WiSec, 2017.

[46] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda, "Curiousdroid: automated user interface interaction for android application analysis sandboxes," in FC, 2016.

[47] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in FSE. ACM, 2012.

[48] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in CCS. ACM, 2014.

[49] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in USENIX Security, 2012.

[50] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in DSN. IEEE, 2009.

[51] acunetix. (2019) Audit your web security with acunetix vulnerability scannerk. Https://www.acunetix.com/vulnerability-scanner/.

[52] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in MoMM. ACM, 2013.

[53] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in WODA & PERTEA. ACM, 2014.

[54] T. Wu and Y. Yang, "Crafting intents to detect icc vulnerabilities of android apps," in CISIS. IEEE, 2016.

[55] H. Feng and K. G. Shin, "Understanding and defending the binder attack surface in android," in ACSAC. ACM, 2016.

[56] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra et al., "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in MobiCom. ACM, 2014.

APPENDIX

A. NON-ENGLISH APPS

In this appendix, we introduce the details in evaluating non-english apps using `TextExerciser`. We select top 5,000 apps from Google Play, filter all the English ones and then randomly select 10 apps as shown in Table X. This selection of apps covers seven different languages ranging from western language like Spanish to languages of east Asia like Japanese. All the apps are popular, i.e., with more than 100,000 downloads.

Table IX
PARAMETER CONFIGURATION OF TEXTEXERCISER'S CNN-RNN CLASSIFIER

#Parameters	#Value	#Parameters	#Value
batch_size	128	12_reg_lambda	0.0
dropout_keep_prob	0.5	max_pool_size	4
embedding_dim	300	non_static	false
evaluate_every	200	num_epochs	1
filter_sizes	3,4,5	num_filters	32
hidden_unit	300		

B. PARAMETER SETTINGS OF TEXTEXERCISER'S CNN-RNN CLASSIFIER

In this appendix, we introduce the parameters adopted in `TextExerciser`'s open-source multi-class CNN-RNN model [27]. After adjusting all the parameters, we find that the default parameters shown in Table IX are still the best and therefore we adopted these default values for `TextExerciser`.

C. DETAILED RESULTS OF TAINTDROID AND RECON

In this appendix, we break down the privacy leaks detected by TaintDroid, Recon and the keyword based approach in Table XI. As mentioned, TaintDroid detects the most number of device identifiers, but is relatively weak in detecting credentials and user inputted locations. As a comparison, both Recon and the keyword based approach detect privacy categories other than device identifiers. One important take-away here is that `TextExerciser` can help all existing tools to detect more privacy leaks.

Table X
NON-ENGLISH APPS COLLECTED FROM GOOGLE PLAY.

#Package Name	#Downloads	Language	Version	Captured Hints	Generated Inputs	Reason for Generation Failure
com.projectm.ezbrother.ssm	500,000+	Korean	1.8.7	3	3	N/A
by.onliner.ab	100,000+	Arabic	1.4.1	3	3	N/A
com.deals.deal	100,000+	Arabic	1.38	2	2	N/A
ru.medsolutions	100,000+	Russian	1.2.3	2	2	N/A
com.moneyforward.android.app	1,000,000+	Japanese	2.0.4	2	2	N/A
de.mobiletrend.lovidoo	500,000+	German	230	1	0	Translation inaccuracy
com.sabqelmfradon	1,000,000+	Japanese	1.1	2	2	N/A
kr.co.dany.threelinediary	500,000+	Korean	2.0.24	2	2	N/A
jp.co.mapple.cotripofficial	100,000+	Japanese	4.1.4	3	3	N/A
jp.co.dwango.nicoch	100,000+	Japanese	1.8.4	1	1	N/A

Table XI
THE CATEGORY OF PRIVATE INFORMATION IN TAINTDROID, RECON AND A KEYWORD-BASED TRAFFIC ANALYSIS IMPLEMENTED BY OURSELVES. WE LIST ALL THE USED KEYWORDS IN THE LAST COLUMN.

Dynamic AnalysisTools	Exerciser	Device Identifier	User Identifier	Contact Information	Location	Credentials	Total
TaintDroid	Monkey	23	0	0	0	0	23
	Monkey+TE	56	0	0	1	0	57
	Stoat	32	0	0	0	0	32
	Stoat+TE	66	0	0	1	0	67
ReCon	Monkey	1	1	0	1	3	6
	Monkey+TE	2	10	0	4	13	29
	Stoat	1	2	0	1	4	8
	Stoat+TE	2	19	0	5	18	44
Keyword Search	Monkey	1	1	0	1	3	6
	Monkey+TE	2	11	1	4	13	31
	Stoat	1	2	0	1	5	9
	Stoat+TE	2	20	1	5	19	47