# Property-Based Testing for Validating User Privacy-Related Functionalities in Social Media Apps

Jingling Sun
University of Electronic Science and Technology of China
Chengdu, China
jingling.sun910@gmail.com

Ting Su*
East China Normal University
Shanghai, China
tsu@sei.ecnu.edu.cn

Jun Sun
Singapore Management University
Singapore, Singapore
sunjunhqq@gmail.com

Jianwen Li
East China Normal University
Shanghai, China
jwli@sei.ecnu.edu.cn

Mengfei Wang
ByteDance
Beijing, China
wangmengfei.pete@bytedance.com

Geguang Pu*
East China Normal University
Shanghai, China
ggpu@sei.ecnu.edu.cn

## ABSTRACT

Social media apps implement many user privacy-related functionalities. For example, TikTok allows users to upload videos that record their daily activities and specify which users can view these videos. Ensuring the correctness of these functionalities is thus crucial. Otherwise, it may threaten the users' privacy or disrupt user experience. Due to the lack of appropriate automated testing techniques, manual testing remains the primary practice for validating these functionalities, which is cumbersome, error-prone, and inadequate. To this end, we adapt property-based testing to validate such functionalities against the properties described by the given privacy specifications. Our key idea is that privacy specifications can be transformed into the Büchi automata, which can (1) determine whether the app has reached unexpected states, *and* (2) guide the testing process. To support the application of our approach, we implemented an automated GUI testing tool, PDTDroid, which can detect the app behaviors that are inconsistent with the privacy specifications. Our evaluation on TikTok, involving 125 real privacy specifications, shows that PDTDroid can efficiently validate privacy-related functionality and reduce manual effort by an average of 95.2% before each app release. Our further experiments on six popular social media apps show the generability and applicability of PDTDroid. PDTDroid has found 22 previously unknown inconsistencies issues in these extensively tested apps (including four user privacy leakage bugs, nine user privacy-related functional bugs, and nine specification issues).

## CCS CONCEPTS

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

*Corresponding author.


*FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil*

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663863

## KEYWORDS

Property-based testing, Android app testing, Non-crashing bugs

**ACM Reference Format:**
Jingling Sun, Ting Su, Jun Sun, Jianwen Li, Mengfei Wang, and Geguang Pu. 2024. Property-Based Testing for Validating User Privacy-Related Functionalities in Social Media Apps. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3663529.3663863

## 1 INTRODUCTION

Social media apps, *e.g.*, TikTok, Facebook, Instagram, and Twitter, are very popular — there are 4.74 billion active social media users worldwide [45]. These apps allow users to share contents (*e.g.*, texts, pictures and videos) and interact with others. Specifically, these apps usually implement several user privacy-related functionalities (*privacy functionalities* for short). For example, a user can manage who can see his or her posted contents, *or*, on the other hand, what contents posted by others can this user see [13]. Ensuring the correctness of such privacy functionalities is important. Because any flaw in these functionalities may negatively affect user experience or even lead to severe privacy issues [18].

**Example.** TikTok [10], a popular social media app (with billions of monthly active users worldwide) developed by our industrial partner ByteDance, implements many privacy functionalities. For example, according to its specifications, one of its privacy functionalities is: "*when* user A is following user B, B is not following A, and B is [set as] a private account, *if* A can see B's public videos *before* B is a private account, *then* A should be still able to see B's public videos." Here, according to the app features, when a (public by default) account is set as private, its original followers can still see its public videos as before (but other accounts trying to follow this account need to be explicitly approved before becoming new followers). Figure 1 illustrates this functionality, where the simplified GUI pages annotated by "User A" and "User B" show the expected behaviors of user A and B, respectively. When A is following B (B is in A's following list, denoted by A1~A2), B is not following A (A is not in B's following list, denoted by B1~B2) and B is a public account (B's private account setting is disabled, denoted by B3~B5), A can see B's public videos (A can see B's videos, denoted by A3~A5). When B is set and keep as a private account (B's

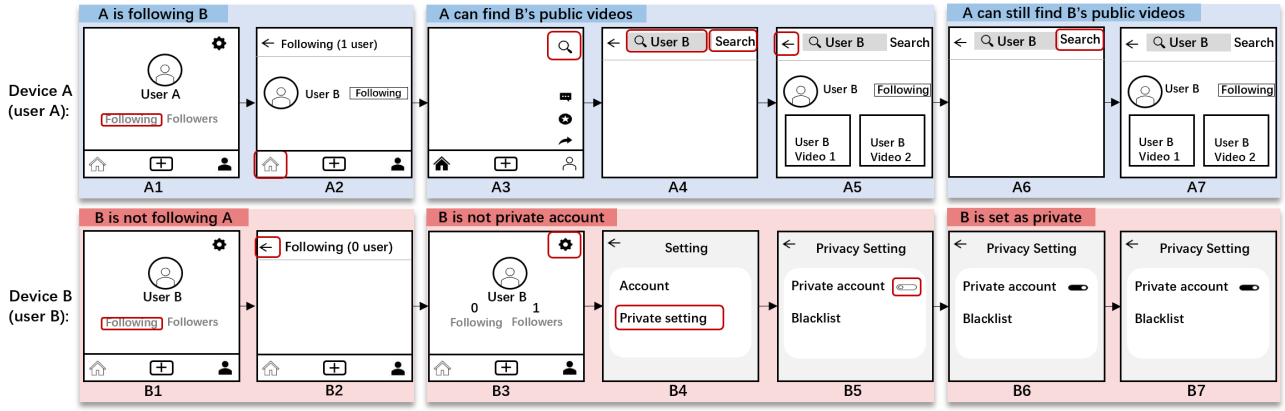


**Figure 1: Illustration of expected app behaviors that are consistent with the sample privacy specification**

private account setting is enabled, denoted by B6~B7), A should be still able to see B's public videos (denoted by A6~A7).

**Difficulties of Current Practice.** Validating such privacy functionalities in practice is non-trivial. For example, many automated GUI testing techniques have been proposed and successfully applied in industry [36, 38, 40], but they are difficult to be applied in this setting. Because they are limited to crashing bugs (we will discuss in detail in Section 5). As a result, the TikTok team follows the most widely-used practice in industry, *i.e.*, *manual testing* [2, 77], to validate these privacy functionalities *w.r.t.* the specifications.

However, manual testing is cumbersome and time-consuming. For example, to check the aforementioned example of privacy functionality, a human tester needs to manually execute these steps: ① logging in the accounts of user A and B on two separate mobile devices, ② posting a public video on the behalf of B, ③ making that A follows B, B does not follow A and B is not private, and then checking whether A can see B's videos, ④ setting B as private, and then checking whether A can still see B's videos. In practice, it may take several minutes to validate such one privacy functionality. However, TikTok has a few hundred pieces of privacy specifications, and one piece of specification usually has to be checked at one time. Moreover, TikTok releases new app versions on a weekly basis. As a result, this manual testing process has to be repeated weekly, which incurs a lot of manual cost.

On the other hand, manual testing could be inadequate due to its small-scale validation, *i.e.*, only checking the *happy* paths without considering other possible adverse conditions. For example, if some privacy issues only manifest when A unfollows B and then follows B again between the steps of ③ and ④, we may miss such issues.

**Our approach and its novelty**. Inspired by the classic idea of property-based testing (PBT) [11], this paper aims to adapt PBT to mitigate the difficulties of current practice in validating privacy functionalities. To apply PBT, one needs to (1) *manually* specify the properties of interest (*e.g.*, usually in the form of assertions), and design data generators to *automatically* generate (random) inputs for property validation [50, 65, 66].

However, instantiating the idea of PBT in our setting is not straightforward. We face two *key* technical challenges: (1) how can we specify the properties of privacy functionalities, which are difficult to be captured by simple assertions? *and* (2) how can we design a data generator to effectively and efficiently validate the privacy functionalities? To tackle these two challenges, we collaborate with the TikTok team and obtain several important observations. First, we observe that (1) the privacy specifications are well-formatted due to their importance, and (2) the privacy functionalities are described in the form of temporal properties (detailed in Section 2.1). Thus, we are inspired to synthesize the properties from the privacy specifications, and represent these properties in the form of linear temporal logic (LTL) formulas [49]. Note that each atomic proposition in such a LTL formula denotes a specific app state (*e.g.*, A is following B), which can be changed or checked by a *privacy operation* (*e.g.*, A is made to follow B).

Second, since LTL formula can be equivalently transformed to Büchi automata [41], we can define the automata-based test adequacy metric (*i.e.*, transition coverage) to guide the data generator, and we also generate random events to exhibit more app states (named as the *property-guided fuzzing* strategy). Specifically, by examining the privacy specifications, we also observe that different privacy functionalities may share similar privacy operations (*e.g.*, posting a public video). Thus, we are inspired to validate *all* privacy functionalities at the same time by choosing appropriate privacy operations to increase transition coverage as quickly as possible (named as the *all-specification checking* strategy). In this way, we can effectively and efficiently validate functionalities.

**Evaluation and Results**. We implemented our approach as a tool named PDTDroid to support validating the privacy functionalities. It uses NLP techniques [42] to synthesize the properties from the specifications in natural language, and outputs any inconsistencies between the specifications and the actual app behaviors (*inconsistency issues* for short). In particular, to automate the validation process, we manually model all the relevant privacy operations in terms of UI events. Note that this is almost an one-time effort (discussed in Section 4.3). Subsequently, PDTDroid automatically explores the app guided by the automata and checks whether all automata have been adequately covered during testing. Once any automaton has reached the accepting state, an inconsistency issue is found and a bug report will be generated.

We applied PDTDroid to TikTok based on 125 pieces of privacy specifications provided by the TikTok team. PDTDroid found eight inconsistency issues (including one privacy functional bug

**Table 1: The syntax patterns used by ByteDance for writing privacy specifications, along with corresponding examples and LTL formula patterns ($c_1$, $c_2$, $c_3$, $c_4$ representing clauses)**

| ID | Syntax Patterns | Samples of Privacy Specifications | LTL Formula Patterns |
|---|---|---|---|
| 1 | **Scene**: any, **Expectation**: $c_1$ | **Scene**: any, **Expectation**: A cannot find B's private video | $G(c_1)$ |
| 2 | **Scene**: $c_1$, **Expectation**: $c_2$ | **Scene**: A is not following B, B is not following A, and B is private, **Expectation**: A cannot find B's public video | $G(c_1 \rightarrow c_2)$ |
| 3 | **Scene**: $c_1$, **Expectation**: **after** $c_2$, $c_3$ | **Scene**: A is following B, and B is following A, **Expectation**: **after** B is blocking A, A is not following B | $G(c_1 \rightarrow X(c_2 \rightarrow c_3))$ |
| 4 | **Scene**: $c_1$, **Expectation**: **if** $c_2$ **before** $c_3$, **then** $c_4$ | **Scene**: A is following B, B is not following A, and B is private, **Expectation**: **if** A can find B's public video **before** B is private, **then** A can find B's public video | $G((\neg c_3 \wedge c_2 \wedge X(c_1 \wedge c_3)) \rightarrow X c_4$ |

and seven specification issues). All these issues have been fixed. Our evaluation also shows that PDTDroid can on average save approximately 95.2% of manual efforts on checking new app versions. Moreover, the optimization strategies (property-guided fuzzing and all-specification checking strategies) indeed contribute to improving the efficiency. Specifically, the two optimizations respectively increase transition coverage by 13.8% and 9.6% within a six-hour testing period. Final, to show the applicability of our approach, we apply PDTDroid on Xigua Video (another social media app from ByteDance), Instagram, Twitter, Bilibili, Threads and Facebook (five social media apps from other app vendors). Based on the 30 pieces of privacy specifications provided by the Xigua Video team and 160 pieces of privacy specifications written by ourselves (based on the privacy settings from the other five apps). PDTDroid found 14 inconsistency issues (including four privacy leakage issues, eight privacy functional bugs and two specification issues) in these six apps. We have reported these issues to the app vendors, all of which have been confirmed and are under fixing. These results show the usefulness of our approach.

To sum up, we make the following contributions:

- We propose a property-based testing approach for validating privacy functionalities, and employ two optimization strategies to improve testing effectiveness and efficiency.
- We implement a GUI testing tool, PDTDroid, to support the application of our approach. PDTDroid can synthesize the properties from the privacy specifications in natural language, and use these properties to find privacy issues.
- We apply PDTDroid to seven popular, well-tested social media apps TikTok, Xigua Video, Instagram, Twitter, Bilibili, Threads, and Facebook. PDTDroid can efficiently validate privacy functionalities of these apps and found 22 previously unknown inconsistency issues.

# 2 BACKGROUND

## 2.1 Privacy Specifications and Functionalities

In this paper, we name the specifications of privacy functionalities as *privacy specifications*. In the TikTok team, the privacy specifications are written in natural language with four typical syntax patterns. In Table 1, column "*Syntax Patterns*" shows these four syntax patterns. In these syntax patterns, "Scene" denotes the scenario where the privacy functionality could be checked, "Expectation" denotes the expectation of the privacy functionality, "if" and "then" denote the condition, and"before" and "after" describe the temporal orders of app states. In Table 1, column "*Samples of Privacy Specifications*" gives some samples of privacy specifications under these syntax patterns. For example, the first specification (with ID 1) in Table 1 stipulates that user A cannot see user B's private videos at any time. If this specification is violated (*i.e.*, A can see the private information of B), it indicates a severe privacy leakage bug (leaking private user data). The fourth specification (with ID 4) in Table 1 is explained in Section 1. If this specification is violated (*i.e.*, A cannot see B's public video), it indicates a privacy-related functional bug.

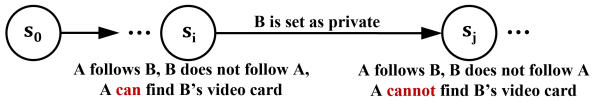


**Figure 2: A state sequence indicating an inconsistency issue**

We observe that each privacy specification requires that some specific (bad) app state should not occur (*i.e.*, safety property). Moreover, the property of privacy functionality can be characterized by a temporal property. Let's take the fourth specification (with ID 4) in Table 1 as an example. As shown in Figure 2, $s_0$ represents the initial app state, while $s_i$ and $s_j$ denote the app state where A follows B, B does not follow A. However, if executing the privacy operation (*i.e.*, B is set as private) on $s_i$, A can no longer find B's public video as before $s_j$. The specification is violated. We can see that this privacy specification can be captured by a temporal property in linear temporal logic formula.

## 2.2 Linear Temporal Logic

Linear temporal logic (LTL) [49] is a temporal logic that describes system behaviors as a sequence of states, extending infinitely into the future. It works with a fixed set of atomic propositions (such as $p_1$, $p_2$, ...). These propositions stand for atomic facts which hold of a system, like "A is following B" and "B is private". A LTL formula has the following syntax given in the Backus Naur form [39]:

$$\phi ::= p|(\neg\phi)|(\phi \wedge \phi)|(\phi \vee \phi)|(\phi \rightarrow \phi)|(X\phi)|(F\phi)|(G\phi)|(\phi U \phi)$$

where, $p$ is an atomic proposition. $\neg$, $\vee$, $\wedge$, and $\rightarrow$ are logical symbols that mean "not", "or", "and", and "implies", respectively. $X$, $F$, $G$ and $U$ are modal operators, which mean "next", "eventually", "globally", and "until", respectively.

Since each privacy specification of the app describes a temporal property, we can formalize these specifications by expressing them as LTL formulas. For example, let $c$ be the clause in a sentence, which may consist of multiple atomic propositions $p$. If a specification has the form of "**Scene**: $c_1$, **Expectation**: **if** $c_2$ **before** $c_3$, **then** $c_4$", it can be represented as "$G((\neg c_3 \wedge c_2 \wedge X(c_1 \wedge c_3)) \rightarrow X c_4)$" (Section 3.1 explains how to do this transformation). The LTL formula means that "Globally, if $c_3$ does not hold and $c_2$ holds, and in the next state both $c_1$ and $c_3$ hold, then $c_4$ should hold in the next state as well", which captures the specification.

## 2.3 Property-based Testing

Property-based testing is a classic testing methodology [11]. The idea is to define a set of properties that the system should satisfy, design a data generator to automatically generate test cases, and checks whether these properties always hold. For example, for the

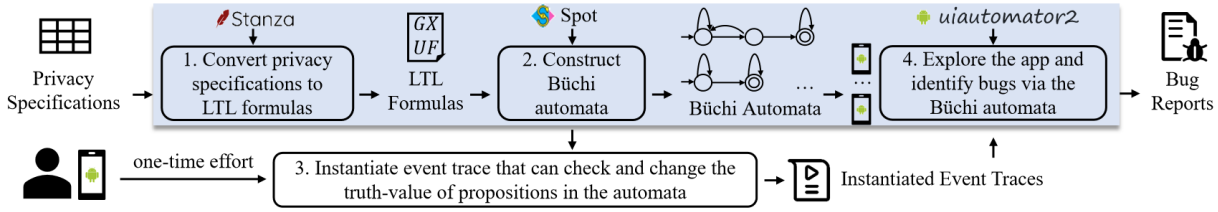


**Figure 3: Workflow of our approach**

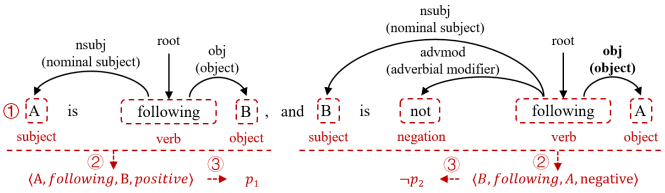


**Figure 4: The example of dependency tree**

property of the privacy functionality discussed in Section 1, we should never generate a sequence of app states that violates the LTL formula $\phi$=G($(\neg c_3 \wedge c_2 \wedge X(c_1 \wedge c_3))\rightarrow$X$c_4$. Thus, given a sequence of app states $\pi$ explored by generated tests, we can check if there exists a prefix of $\pi$ matching $\neg\phi$. If such a prefix is found, it indicates that the app violates the property $\phi$.

## 3 APPROACH

At a high level, our approach takes an app under test (AUT) and some privacy specifications $\mathcal{P}$ as inputs, and outputs a set of the counterexamples, *i.e.*, a series of GUI events that can reach an app state where a specification in $\mathcal{P}$ is violated. To do this, our approach works in four steps as shown in Figure 3. First, it automatically converts the input privacy specifications into LTL formulas (Section 3.1). Second, to facilitate testing, each event trace that can check and change the truth value of atomic proposition in LTL formulas is instantiated (Section 3.3). Third, the LTL formulas are systematically transformed into Büchi automata (Section 3.2). Finally, it automatically performs oracle checking and explores the AUT based on the automata (Section 3.4). Next, we will present the formulation and technical details of our approach.

### 3.1 LTL Formula Generation

Our approach starts by transforming privacy specifications into corresponding LTL formulas, which involves two phases:

**(1)** First, our approach uses four transformation rules (as shown in Table 1) to transform the natural language specification into the corresponding LTL formula automatically.

**Transformation rules.** For each syntax pattern in Table 1, we have devised a corresponding pattern of LTL formula, displayed in the "LTL Formula Patterns" column of Table 1. We refer to the correspondence between natural language syntax patterns and LTL formula patterns as transformation rules.

Based on these manually formulated transformation rules, we use regular expression matching to check whether any natural language specification conforms to a certain syntax pattern, and if so, we convert it into the corresponding LTL formula.

***Example***. One of the transformation rules is that if the sentence has the form of "**Scene**: $c_1$, **Expectation**: **if** $c_2$ **before** $c_3$, **then** $c_4$", it can be converted into an LTL formula with the form of G($(\neg c_3 \wedge c_2 \wedge X(c_1 \wedge c_3))\rightarrow$X$c_4$). Based on this rule, we can convert the privacy specification with ID 4 in Table 1 to "G(¬(B is private) ∧ (A can find B's video card) ∧ $X$((A is following B, B is not following A, and B is private) ∧ (B is private)) → $X$(A can find B's video card))".

**(2)** Next, our approach analyzes each clause in the LTL formulas, decomposes it into conjunctions of propositions, and replaces them with symbols automatically, as detailed below:

① We use Stanza [68]'s dependency analysis to identify the main components (i.e. subject, verb, object, and negation) of the clause. The output of dependency analysis is a dependency tree, which can be exploited to identify the main components of a sentence. Figure 4 shows the result of the Stanza dependency analysis on the sentence "A is following B, and B is not following A". Note that only arrows that determine the main components of this sentence are shown here. There is an arrow pointing from the third word "following" to the first word "A" in Figure 4. The arrow is labeled "*nsubj*", indicating that the word pointed by the arrow is the subject ("A" here) of the clause, and the starting word of the arrow is the verb ("following" here) of the clause. Similarly, the arrow marked with "*obj*" can be used to identify the object of the clause, while the arrow marked with "*advmod*" starting from the verb can be used to identify the negation of a clause.

② We represent each main component set as a proposition, denoted by a tuple $\langle s, v, o, f\rangle$. Here, $s$ is the subject, $v$ is the verb, $o$ is the object, and $f$ is a flag that represents whether the proposition expresses a negative meaning. The value of $f$ is negative if a verb-dependent negation is present, otherwise positive. As an example, based on the result of dependency analysis, the clause in Figure 4 is composed of two propositions ⟨A, following, B, positive⟩ and ⟨B, following, A, negative⟩.

③ We use the symbol $p$ to represent each proposition and replace the propositions with these symbols in order as $p_1$, $p_2$, and so on. In particular, if a proposition has the same subject-verb-object component as another proposition, we use the same symbol to represent them. Likewise, we merge proposition phrases that have the same subject-verb-object but are marked as negative. For example, if ⟨A, following, B, positive⟩ is replaced by $p_1$, ⟨A, following, B, negative⟩ can be replaced by $\neg p_1$.

Through the preceding three-step conversion process, we convert all privacy specifications into LTL formulas in final form. These LTL formulas are composed of temporal logic symbols and the least types of symbols representing propositions.

***Example***. The privacy specification with ID 4 in Table 1 will ultimately be translated into $G(\neg p_3 \wedge p_4 \wedge X(p_1 \wedge \neg p_2 \wedge p_3) \rightarrow Xp_4)$, where $p_1$, $p_2$, $p_3$, and $p_4$ respectively represent "A is following B", "B is following A", "B is private" and "A can find B's public video".

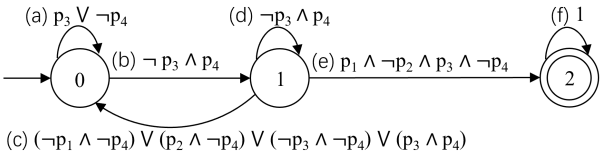


**Figure 5: Automaton corresponding to $\neg G(\neg p_3 \wedge p_4 \wedge X(p_1 \wedge \neg p_2 \wedge p_3) \rightarrow X p_4)$ for checking the fourth privacy specification in Table 1.**

## 3.2 Büchi Automaton Construction

For each LTL formula $\phi$, we use a standard approach [20] to construct a Büchi automaton $a$ accepts paths satisfying $\neg\phi$, which will be used for automatic oracle checking in Section 3.4. The automaton $a$ is defined as a tuple $a = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, $\Sigma$ is a finite set of symbols called alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function with each transition $\langle q, \sigma, q' \rangle \in \delta$ denoting a state transition from $q$ to $q'$ triggered by satisfying the condition $\sigma$, $q_0$ is the initial state, and $F \subseteq Q$ is a set of accepting states. Among others, $\Sigma$ consists of boolean expressions over the propositions, $F$ denotes either a compliant state or an undesired state, depending on the kind of property. In our testing scenario, all the privacy specifications describe safety properties, and thus all accepting states of the automaton denote the undesired app states. Specifically, to facilitate automatic state transitions, the generated automaton is represented as a deterministic finite automaton (DFA). In our setting, such a DFA does not lead to excessive states and transitions (in our experiment, the average number of states and transitions of the automata synthesized from privacy specifications are 2 and 5, respectively).

***Example***. The negation of the LTL formula $G(\neg p_3 \wedge p_4 \wedge X(p_1 \wedge \neg p_2 \wedge p_3) \rightarrow X p_4)$ is $\neg G(\neg p_3 \wedge p_4 \wedge X(p_1 \wedge \neg p_2 \wedge p_3) \rightarrow X p_4)$. The automaton converted from this formula is illustrated in Figure 5.

## 3.3 Privacy Operation Instantiation

Definition 1. ***GUI Page, GUI Widget, GUI Event, and GUI Event Trace***. *An Android app is a GUI-centered event-driven program. Each of its GUI pages is a runtime GUI layout $\ell$, i.e., a GUI tree $T$. Each node of this tree is a GUI widget (or view) $w$ ($w \in \ell$). Specifically, each widget $w$ has some attributes, e.g., className (i.e., the widget type), resourceId (i.e., the widget id), and text (i.e., the widget text). A GUI event $e = \langle t, w, d \rangle$ is a tuple, where $e.t$ denotes the event type (*e.g., `click`, `edit`*), $e.w$ denotes the target GUI widget of the event, and $e.d$ denotes the optional data associated with $e$ (e.g., a string/number for edit). A GUI event trace is a sequence of GUI events $E = [e_1, e_2, \ldots]$. $E$ can be executed on the AUT to obtain a sequence of GUI layouts (pages) $L = [\ell_1, \ell_2, \ldots]$.*

Definition 2. ***Proposition, Controllable Proposition, Uncontrollable Proposition, Privacy Operation, and App State***. *The proposition is the statement that describes the specific state of the target system (AUT here), denoted by $p$. When the AUT is used by users, whether the truth value of a proposition $p$ is $True$ or $False$ may change at any time, which can be observed on the specific page by the specific user. We refer to the proposition whose truth value can be deterministically controlled by executing certain UI events as controllable propositions and whose truth value that cannot be controlled by users as uncontrollable propositions. A privacy operation is defined as a GUI event trace which has the capability to change or check the truth value of some proposition at runtime. Let $P_A = \{p_0, p_1, \ldots\}$ be the set containing all propositions from the automata set $A$ ($A$ contains all the automata transformed from the privacy specifications). Specifically, we use $P_A = CP_A \cup UCP_A$ to denote the sets of controllable propositions $CP_A$ and uncontrollable propositions $UCP_A$ in automata set $A$, respectively. As a result, the app state can be characterized by the truth values of these propositions in $P_A$, represented as $s = [v_{p_0}, v_{p_1}, \ldots]$, where each $v_{p_i}$ denotes the truth value ($True$ or $False$) of the corresponding proposition $p_i$.*

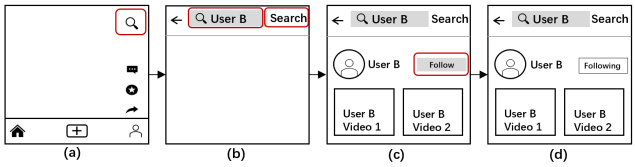


**Figure 6: Privacy operation to change the truth value of $p_1$**

***Example***. In the example mentioned at the end of Section 3.1, the proposition $p_3$ ="B is private" denotes the account status of a user which can be changed by some privacy operation. Thus, $p_3$ is a controllable proposition. The proposition $p_4$ = "A can find B public video" denotes the relationship between two users of the app. This relationship cannot be explicitly changed by users via some privacy operation. Thus, $p_4$ is an uncontrollable proposition. If the privacy specifications to be checked only involve these two propositions, the app state can be characterized as $\{v_{p_3} = True, v_{p_4} = False\}$.

In the previous steps, the privacy specifications written in natural language are converted into the corresponding automata. When the truth values of the propositions change, the states of these automata will change. To control and check the transitions of the automaton, (1) for each controllable proposition, we manually define how to change its truth value, and (2) for each controllable or uncontrollable proposition, we manually define how to check its truth value. First, for each controllable proposition, we manually instantiate a privacy operation $E_{control} = [e_{control_1}, e_{control_2}, \ldots]$ to change its truth value. Once $E_{control}$ is executed from the starting GUI page of AUT, the truth value of the target proposition changes. Second, for each controllable or uncontrollable proposition, we identify a key GUI widget $w_{check}$ and instantiate a privacy operation $E_{check} = [e_{check_1}, e_{check_2}, \ldots]$ to check its truth value. When $E_{check}$ is executed from the starting GUI page of the AUT, the AUT can reach a GUI page on which we can check the status of $w_{check}$ to determine the truth value of the target proposition.

Note that automatically constructing the traces of privacy operations through their descriptions is challenging in practice, as it is difficult to accurately infer the GUI pages on which these operations can be performed. Manually constructing the trace is a one-time effort and typically acceptable in practice as (1) we need at most two traces (*i.e.*, $E_{control}$ and $E_{check}$) for each proposition and (2) a tester usually already constructs these traces during manual testing. We assess the required manual effort in Section 4.4.

***Example***. The event trace that can change the truth value of proposition $p_1$ ("A is following B") is illustrated by the four red boxes in Figure 6. The trace consists of four events (*i.e.*, clicking the *search* button, entering the user name of B, clicking the *search* button, and

**Algorithm 1:** Identify inconsistency issues via automata

```
Inputs : A: set of automaton converted from privacy specifications,
         CP_A: set of controllable propositions involved in A,
         UCP_A: set of uncontrollable propositions involved in A,
Output : bugsInfo: information of detected bugs
1  Function Main:
2      foreach automaton a ∈ A do
3          a.q_c ← a.q_0;
4      bugsInfo ← ∅; eventTrace ← [];
5      while not timeout do
6          currentAppState ← checkPropositions(UCP_A ∪ CP_A);
7          foreach automaton a ∈ A do
8              foreach t ∈ a.δ do
9                  if checkTransition(a, t, currentAppState) then
10                     a.makeStateTransition();
11             if a.q_c ∈ a.F then
12                 bugsInfo ← bugsInfo ∪ (eventTrace, a);
13         eventTrace ←
             explore(currentAppState, A, CP_A, eventTrace);
14     return bugsInfo
```

**Algorithm 2:** Explore the app guided by automata

```
1  Function explore(currentAppState,automata,CP_A,eventTrace):
2      candidateTransitions ← getPossibleTransitions(automata);
3      maxScore ← 0;
4      candidateAppStates ←
         getPossibleNextState(currentAppState, CP_A);
5      foreach s ∈ candidateAppStates do
6          score ← 0;
7          foreach t ∈ candidateTransitions do
8              if checkWeakSatisfy(t.σ, s) then
9                  score ← score + t.weight;
10         if score > maxScore then
11             selectedAppStates ← s;
12             maxScore ← score;
13     changeEvents ←
         getEvents(currentAppState, selectedAppStates);
14     eventTrace ← eventTrace :: executeEvents(changeEvents);
15     eventTrace ← eventTrace :: executeRandomEvents();
16     return eventTrace;
```

clicking the *follow* button). After these four events are executed by user A, the truth value of $p_1$ can change from $False$ to $True$ (or from $True$ to $False$). On the other hand, the three red boxes in the first two pages in Figure 6 illustrate the steps to check the proposition $p_1$. On page (c), if user B's user card contains a button with the text "following", the truth value of $p_1$ is $True$ (*i.e.*, A is following B), otherwise the truth value of $p_1$ is $False$ (A is not following B).

## 3.4 Inconsistency Issue Identification

After deriving the Büchi automata and privacy operations, our approach explores the AUT and checks whether there is any inconsistency between the specification and the app behaviors. In particular, we propose the all-specification checking and property-guided fuzzing strategies to validate privacy functionalities.

Algorithm 1 shows how our approach can automatically perform oracle checking to detect inconsistency issues. We use the same definition of automaton as in section 3.2 to explain our algorithm, with $a, Q, \Sigma, \delta, q_0$, and $F$ maintaining their meanings. In addition, we introduce a new variable $q_c$ for each automaton to record the current state of the automaton. At the beginning of the algorithm, it first initializes each automaton's variable $q_c$ to the initial state $q_0$ (lines 2-3). Then, it initializes $bugsInfo$ and $eventTrace$ to record all detected bugs and all executed events, respectively (line 4). In the main loop (lines 5-13), the algorithm continually repeats the following three steps: (1) It checks the truth values of all propositions involved in all automata by *checkPropositions* and saves the result into $currentAppState$ (line 6). (2) It traverses each automaton in turn and checks whether a transition has occurred (lines 7-10). In detail, each automaton reviews all its transitions $t = \langle q, \sigma, q' \rangle \in \delta$ (line 8), uses $checkTransition$ to confirm ① if the transition starts from the current state ($q_c$) and ② if its transition condition $\sigma$ is $True$, and returns $True$ when both are met (line 9). If the return value of $checkTransition$ is $True$, the automaton will transition to state $q'$ and update its $q_c$ accordingly via function $makeStateTransition$ (line 10). It is worth noting that since each automaton generated by Section 3.2 is represented as a DFA. Thus, at most one transition of each automaton can be satisfied at any given time. (3) It checks whether any automaton has reached the accepting state (line 11). If so, it indicates that some specification is violated, and records the violated specification and the previously executed events (line 12). Finally, the algorithm executes some events that may change the app state by $explore$ (the strategy for executing events will be introduced in the Algorithm 2) (line 13). Through the above loop, the algorithm changes the app state and simultaneously checks all privacy specifications. We term this strategy as the *all-specification checking* strategy. Eventually, the algorithm returns any found inconsistency issues during testing (line 14).

Note that Algorithm 1 solves the oracle checking problem. Algorithm 2 focuses on exploring the app effectively, which (1) utilizes all automata to guide test case generation with the aim of improving the transition coverage, and (2) executes random events to explore more possible app states. To guide the generation of test cases, a variable $weight$ is added to each transition. Every time a transition is covered, its $weight$ will be decreased. When the function $makeStateTransition$ on line 9 of Algorithm 1 is called, the $weight$ of the corresponding transition will also be updated. In our experiment, we set the $weight$ to one-tenth during each update. If efficiency is prioritized, the denominator can be increased, whereas if sufficiency is emphasized more, the numerator can be increased.

Algorithm 2 first uses $getPossibleTransitions$ to find any transition of all automata that may occur (*i.e.*, any transition of all automata whose start states are $q_c$) and stores them in a list named $candidateTransitions$ (line 2). Afterwards, the algorithm looks for a feasible controllable proposition that, when modified, can cover as many transitions as possible while maximizing the sum of the weights of the satisfied transitions (lines 4-12). We term this strategy as the *property-guided fuzzing* strategy. Note that we may not always be able to achieve the transition. In particular, some transitions on the automaton may never be covered because there is no corresponding bug in the app (*e.g.*, transitions (e) and (f) in Figure 5). To achieve this, the algorithm employs $getPossibleNextState$ to identify all possible app states resulting from altering each controllable proposition in the current state and records these states in $candidateAppStates$ (line 4). For example, assuming that the automata set $A$ involves only two controllable propositions, $p_1$ and $p_2$, and the current app state is $[v_{p_1} = True, v_{p_2} = True]$. Then, by

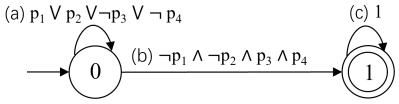


**Figure 7: Automaton corresponding to $\neg G(\neg p_1 \wedge \neg p_2 \wedge p_3 \rightarrow \neg p_4)$ for checking the second privacy specification in Table 1.**

changing one controllable proposition, the app state at the next moment may be $[v_{p_1} = False, v_{p_2} = True]$ or $[v_{p_1} = True, v_{p_2} = False]$. We store these two app states in the list *candidateAppStates*. Next, for each candidate app state, it calculates which candidate transition' condition can be weakly satisfied (*i.e.*, assuming the transition can be enabled if the corresponding controllable propositions hold) under the current app state by $checkWeakSatisfy$. If so, it adds the weight of the transition to the score corresponding to the candidate app state, and selects the app state with the highest score as the last selected one (lines 5-12). Finally, the algorithm uses *getEvents* to identify the controllable propositions that need to be changed to reach the target app state and returns the event traces required to make these changes. (line 13). Next, the algorithm executes these events sequentially and records the events that have been successfully executed by *executeEvents* (line 14). In particular, the algorithm also randomly executes some events by *executeRandomEvents* (with the chance of coin-flipping) to reach more diverse app states (line 15). In this random execution, the algorithm randomly selects a GUI widget from the current GUI page and randomly selects an event type to generate a GUI event for that widget and executes that event. Note that we avoid repeating a small number of the same event traces by considering all the automata at once, and dynamically updating the transition weights.

***Example***. Assuming only the second and fourth privacy specifications in Table 1 are needed to be checked (corresponding to the automata shown in Figure 7 and Figure 5 respectively) and the initial app state is that A is not following B, B is not following A, B is private and A can find B public video. (1) Algorithm 1: First, two automata all initiate at state 0 (line 2). Subsequently, after the execution of line 5, $p_1$, $p_2$, and $p_3$ are evaluated as $False$, and $p_4$ are evaluated as $True$. Then, after line 9 is executed, the automaton in Figure 5 migrates to state 1 through transition (b), and the automaton in Figure 7 migrates back to state 0 through transition (a). (2) Algorithm 2: To enhance transition coverage, Algorithm 2 calculates, through the execution of lines 2-12, that proposition $p_3$ should be changed. As this change could lead to covering transition (c) of the automaton in Figure 5 and transition (b) of the automaton in Figure 7. On the other hand, changing either $p_1$ or $p_2$ at this point would not cover these two transitions. To this end, it changes the truth value of $p_3$ (lines 13-14), performs random events (line 15), and then continues the next iteration.

# 4 EVALUATION

Our experiment aims to answer these research questions:

- **RQ1** : How effective is PDTDroid in validating privacy functionalities of TikTok? Can PDTDroid find inconsistency issues?
- **RQ2** : How do the optimization strategies in PDTDroid perform in improving the testing effectiveness and efficiency?
- **RQ3** : Can our property-based testing approach (supported by PDTDroid) be applied to validate other social media apps?

## 4.1 Tool Implementation

We implemented a prototype tool PDTDroid to support our approach. Given an app and its privacy specifications (written in the syntax patterns illustrated in Table 1) as input, PDTDroid outputs any found inconsistency issues (including the issue-triggering traces). To convert the privacy specifications to LTL formulas, we use Stanford's Stanza [68] for dependency analysis. We use Spot libraries [67] to obtain Büchi automata from LTL formulas. We use WEditor [75] to help record event traces. We use the UI Automator test framework [69] to execute events and dump GUI layouts. We have made PDTDroid publicly available at *https://github.com/property-driven-privacy-testing/home*.

## 4.2 Experiment Setup

*4.2.1 Experiment Setup of RQ1.* The TikTok team provided us a total of 125 privacy specifications in natural language. To answer RQ1, we used PDTDroid to automatically convert all these privacy specifications to LTL formulas. After that, we manually checked the correctness of these LTL formulas (Section 4.4 discusses the conversion accuracy). We invited one human tester from the TikTok team to help us instantiate the corresponding event traces for the propositions involved in these LTL formulas. The tester manually instantiated a total of 54 event traces for the 125 privacy specifications (Section 4.4 discusses the manual cost). We tested one recent version of TikTok (version 25.1.0) on real Android devices (OPPO A11s, Android 10.0) and allocated 6 machine hours for privacy testing. We manually inspected all the reported inconsistency issues to find the unique ones based on the issue-triggering traces. We reported all the found issues to the TikTok team for confirmation.

*4.2.2 Experiment Setup of RQ2.* We conducted an ablation study to evaluate whether the optimization strategies can improve testing effectiveness and efficiency. Specifically, we built two baselines:

- Baseline A: *This baseline evaluates how the all-specification checking strategy performs in improving testing efficiency.* The only difference between Baseline A and PDTDroid is the strategy of checking the privacy specifications. Baseline A checks one specification at one time, while PDTDroid checks all the specifications at the same time. Note that both Baseline A and PDTDroid use the property-guided fuzzing strategy to generate tests.
- Baseline B: *This baseline evaluates how the property-guided fuzzing strategy performs in improving testing effectiveness and efficiency.* The only difference between Baseline B and PDTDroid is the strategy of generating tests. Baseline B mimics the state-of-the-practice GUI testing tool, Monkey [40], to generate random tests (*i.e.*, randomly exploring GUI pages), while PDTDroid uses the automata to guide test generation for improving transition coverage. Note that both Baseline B and PDTDroid use the all-specification checking strategy.

To compare with PDTDroid, we allocated 6 hours for these two baselines based on the same set of 125 privacy specifications. We measured the transition coverage of all automata and counted the number of detected inconsistency issues.

*4.2.3 Experiment Setup of RQ3.* To address RQ4, we chose the six top downloaded apps in the category of social media apps on Google Play Store [23] and Xiaomi GetApps [79]. They are Xigua Video

**Table 2: List of eight inconsistency issues found by PDTDroid in TikTok**

| ID | Scenes | Expectations | Actual behaviors |
|---|---|---|---|
| 1 | A is blocking B | B's video should not appear on A's recommendation page. | B's video will appear on A's recommendation page. |
| 2 | B has a posted private video | B can find B's private video under comprehensive search. | B cannot find B's private video under comprehensive search. |
| 3 | B has a posted private video | B can find B's private video under video search. | B cannot find B's private video under video search. |
| 4 | A is blocking B | A can mention B on the release page, but B cannot mention A on the release page. | A cannot mention B on the release page. |
| 5 | A is blocking B | A can mention B on the comment page, but B cannot mention A on the comment page. | A cannot mention B on the comment page. |
| 6 | B is private | B can find B's user card under user search, and there is an icon indicating that B is private. | B can find B's user card under user search, but there is no icon indicating that B is private. |
| 7 | A is following B, B is following A, B is private | A can find B's user card under user search, and there is an icon indicating that B is private. | A can find B's user card under user search, but there is no icon indicating that B is private. |
| 8 | B is private, B has a posted video | B can find B's video under video search. | B cannot find B's video under video search. |

(version 7.3.4, 1B+ installations), Instagram (version 272.0.0, 5B+ installations), Twitter (version 9.98.0, 1B+ installations), Bilibili (version 7.45.0, 1B+ installations), Threads (version 300.0.0, 100M+ installations) and Facebook (version 426.0.0, 5B+ installations). Among them, Xigua Video is developed by ByteDance, and we tested it based on 30 privacy specifications provided by its team. The other five apps are developed by other app vendors. Since these selected apps are social media apps, and thus they have some similar privacy functionalities of TikTok. We could migrate the privacy specifications from TikTok to these apps. We carefully checked whether each privacy specification from TikTok is relevant to the selected apps according to their privacy features, settings and prompts. For example, when disabling Instagram's "allow photos remix", Instagram will prompt that "No one can create new remixes with your photos. Existing remixes will not be affected unless you delete your photos or block the user." Based on this privacy setting and its prompt, we constructed a privacy specification: "Scene: A disables 'allow photos remix', Expectation: B cannot create new remixes with A's photos." In this process, we used the four syntax patterns in Table 1 to construct the privacy specifications for these selected apps. We find that these four syntax patterns are comprehensive enough to capture all the observed privacy functionalities in these apps. However, we also find that some of TikTok's privacy functionalities do not exist in other apps (*e.g.*, Bilibili cannot set user accounts as private), and thus some privacy specifications cannot be migrated. Additionally, since we are not the app vendors, we can only construct the privacy specifications according to the publicly available information of these selected apps. Finally, we constructed 35, 25, 40, 30 and 30 privacy specifications for Instagram, Twitter, Bilibili, Threads and Facebook, respectively. We allocated 24 hours for testing each of these apps. Once the privacy specifications are constructed, applying PDTDroid to these new apps is straightforward. The only manual task is to instantiate the event traces for the LTL formulas.

## 4.3 Experiment Results

*4.3.1 Results for RQ1.* Table 2 shows the inconsistency issues found by PDTDroid. The second and third columns list the scenes and expectations described in the privacy specifications, while the fourth column provides the actual behaviors of the app observed by PDTDroid. After we reported the inconsistency issues, the developers confirmed that the first issue (the issue with ID 1 in Table 2) is a privacy functional bug in TikTok. Due to the algorithmic randomness of the recommendation functionality, finding this bug requires multiple checks of the related privacy specification. As a result, this bug has not been found by manual testing. For the remaining seven inconsistency issues, the developers responded

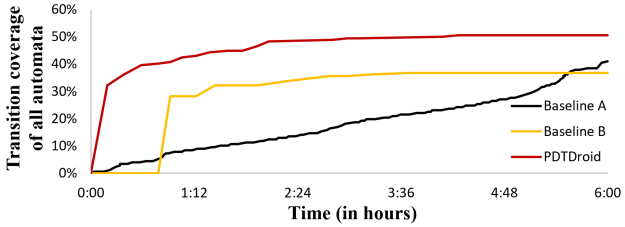


**Figure 8: Comparison between Baseline A, B, and PDTDroid in terms of transition coverage of all automata in six hours.**

that the corresponding privacy specifications had not been updated timely after updating the functionalities of TikTok. These issue reports are still crucial for TikTok as the testing process for privacy functionalities requires referencing to these specifications. So far, all these eight issues have been fixed. Note that our approach does not incur any false positives. We interviewed five testers of the TikTok team. Among them, two are in charge of manually testing privacy functionalities, while the others are responsible for developing automated testing tools. They consistently gave positive feedback that PDTDroid is useful because it can help them check privacy functionalities and spot the specifications issues.

*4.3.2 Results for RQ2.* Figure 8 compares the performance of Baseline A, B, and PDTDroid in terms of transition coverage in validating privacy functionalities. Baseline A, B and PDTDroid are denoted by the black, yellow and red curves, respectively. The horizontal axis denotes the 6-hour testing time, and the vertical axis denotes the transition coverage of of all the automata during testing, respectively. Note that if the app does not violate some specification, some transitions of the corresponding automaton will never be covered. For example, for the automaton in Figure 5, the transitions (e) and (f) can never be covered if the corresponding specification is never violated. The maximal transition coverage of this automaton is 4/6≈66.6%. Therefore, the transition coverage of Baseline A, B, and PDTDroid cannot always reach 100%.

From Figure 8 , we can see that PDTDroid performs better than both Baseline A and B. Benefiting from both the property-guided fuzzing and all-specification checking strategies, PDTDroid increases the transition coverage much faster than the two baselines. PDTDroid also finally achieved the highest transition coverage because it can reach more diverse app states when checking all the specifications simultaneously. However, Baseline A may not be able to sufficiently test each specification with diverse app states due to its inefficiency, while randomness of Baselines B may prevent it from reaching the corresponding scenes for some privacy specifications. Specifically, PDTDroid achieved a respective increase of 13.8% and 9.6% in transition coverage compared to Baselines A and

**Table 3: Inconsistency issues found by PDTDroid in Xigua Video, Instagram, Twitter, Bilibili, Threads and Facebook**

| Type | ID | App | Issue Description |
|---|---|---|---|
| Privacy Leakage | 1 | Instagram | B can see A's deleted videos. |
| | 2 | Twitter | B can see A's deleted videos. |
| | 3 | Twitter | After A removes B from circles, B can still see A's "circle-only" tweets. |
| | 4 | Bilibili | The content that is not visible to other users (such as the name of a video published by a private account) can still be seen via the UI layout files. |
| Privacy Functional Bugs | 5 | Instagram | After B hides the story from A, A sometimes does not appear in B's block list. |
| | 6 | Instagram | Sometimes the number of followers displayed on the user's homepage differs from the number of followers displayed in the followers list. |
| | 7 | Instagram | After B hides the story from A, B cannot see A's story on A's homepage, but can see A's story on the recommendation page. |
| | 8 | Twitter | Crash when user sets visible topic in the setting page. |
| | 9 | Bilibili | After modifying the privacy settings multiple times, the user cannot set the visibility of the fan list unless the app is reinstalled. |
| | 10 | Facebook | After frequently accessing the privacy settings, the user will not be able to open the settings page unless the app is reinstalled. |
| | 11 | Facebook | After A unfollows B, B is still in A's friends list. |
| | 12 | Threads | After turning on the replies filtering and specifying the hidden words, users can still see the replies from other users that include those hidden words. |
| Specification Issues | 13 | Xigua Video | After A blocks B, A cannot search for B. |
| | 14 | Xigua Video | B cannot search for his private video. |

B within a six-hour testing period. On the other hand, PDTDroid found 8 inconsistency issues, while Baseline A and B found 7 and 6 issues, respectively. Both Baseline A and B did not found the issue with ID 1 (which is a real functional bug) in Table 2. Thus, the results show that the two optimization strategies of PDTDroid can help improve testing effectiveness and efficiency.

*4.3.3 Results for RQ3.* Based on the constructed privacy specifications for the selected apps, PDTDroid has successfully found 14 inconsistency issues, which affect the latest versions of these apps. Table 3 gives the detailed information of these found issues. (1) The first four issues (with issue ID 1∼4) are privacy leakages, which could be dangerous for users. For example, the first one indicates that if A (a user of Instagram) accidentally posted a video that reveals A's privacy, and immediately deleted it, other users can still see this deleted video within half one hour. This issue is difficult to be found by manual testing due to its specific issue-triggering condition: it only manifests when A has one draft video and two public videos in A's video list. For another example, the fourth issue indicates that if user A sets itself as a private account, user B can still see A's private videos via the dumped GUI information. This issue is hard to be found by manual testing as the information is invisible from normal GUI pages. (2) The fifth to twelfth issues (with issue ID 5∼12) are functional bugs of the privacy functionalities that may not compromise users' privacy but could negatively affect user experience. For instance, some privacy settings in some apps (Bilibili and Facebook) are buggy: changing the privacy settings for multiple times breaks the settings (issue ID 9 and 10). These issues may be difficult to be found by manual testing. Because human testers typically check each privacy functionality only once due to the limited testing budgets. The issues with ID 5 and 6 are sporadic, and they are only triggered by multiple executions of the same privacy functionalities. The issue with ID 7 also requires multiple executions of the privacy functionality due to the randomness of the recommendation functionality. (3) The last two issues in Xigua Video are caused by the outdated specifications. We have reported all these found issues to the app vendors, all of which have been confirmed and under fixing. These results indicate that our property-based testing approach is general and applicable to different social media apps.

## 4.4 Discussion & Lessons Learned

In this section, we discuss lessons learned and practical experience obtained from this work.

**PDTDroid can help save the manual cost in practice.** In RQ1, the first instantiation on TikTok (version 25.1.0) took 8 person-hours for all the 125 privacy specifications, averaging 3.8 minutes per specification. On the other hand, the human tester reports that the cost needed by manually testing all the privacy specification without PDTDroid is about 6 hours. However, to instantiate PDTDroid on a new app version, a human tester only needs to check whether the event traces from a prior version are valid (*e.g.*, whether the target UI widgets of these event traces have changed). To investigates the manual cost of using PDTDroid in practice, we invited one human tester to apply PDTDroid on five new app versions (26.1.0, 27.1.0, 28.1.0, 28.6.0 and 28.7.0) based on the prior version (25.1.0) tested in RQ1. Specifically, the tester automatically replayed the event traces instantiated for a prior app version on this new app version, *and* manually updated any failed event traces. We recorded (1) the number of failed event traces due to app updates, and (2) the time cost of manually fixing the failed event traces. The statistics show that it takes an average of 0.29 person-hour to test a new app version with PDTDroid. It saves about 95.2% (≈(6-0.29)/6) of the time cost of manual testing. Specifically, the numbers of failed event traces for versions 26.1.0, 27.1.0, 28.1.0, 28.6.0, and 28.7.0 were only 7, 6, 3, 4, and 3, respectively. It indicates that the human testers can increasingly benefit from PDTDroid in reducing the manual testing time when validating new app versions.

**Lightweight natural language processing technique can effectively synthesize the privacy specification.** As shown in Section 3.1, we transform the privacy specification into the corresponding LTL formula by leveraging customized transformation rules and dependency analysis. In our experiment, this approach can achieve high efficiency and accuracy. Specifically, PDTDroid only takes an average of seconds to complete the conversion of each privacy specification. The conversion accuracy (284/315≈90.2%) is also high. Only 31 out of 315 LTL formulas were converted inaccurately. We found that all conversion errors are caused by inaccurate dependency analysis results from the NLP tool Stanza we used, these conversion errors will be found when we instantiate the proposition-related event trace for each proposition (step 2 in Figure 3). For example, the propositions that are missing the object can be noticed and easily fixed by anyone with basic language skills.

**Threats to validity** The primary threat to external validity for this study involves the representativeness of our app and the corresponding privacy specifications for inspection. However, we do reduce this threat to some extent by selecting one of the most popular social apps, TikTok as the major subject, which has complex

user privacy-related functions. Moreover, our testing of TikTok is based on privacy specifications provided by ByteDance, which are used in the real industrial setting.

## 5 RELATED WORK

**Privacy leakage detection in Android apps.** Several methods have been proposed for detecting privacy leakage in Android apps. Some work [3, 22, 32, 37, 76] identifies potential privacy leakage by analyzing the source code or bytecode of Android apps. However, these methods are limited by their inability to account for the dynamic behavior of an app, so they *cannot* detect the privacy leakage related to specific app states, which are only exposed during dynamic execution. As for meta-information analysis, there are some work [28, 44, 56, 78, 83] uses machine learning to detect Android malware (including Adware, Ransomware, Scareware, and SMS Malware) that maliciously leaks user privacy. These work focuses on identifying malicious apps, not accidental privacy-related bugs in regular apps. Some work [5, 16, 33, 54] combines dynamic and static techniques to detect privacy leaks in Android apps. However, the testing purposes of these work are different from our approach. For example, COVERT is a tool for compositional analysis of Android inter-app vulnerabilities. Some work [9, 25, 27, 57, 74, 82] targets privacy policies (describing what information the app will collect), which are different from the privacy specifications (describing the expected behaviors of privacy functionalities) used in our work. Overall, existing privacy leakage detection techniques *cannot* detect the bug concerned in this paper. Moreover, as there are no benchmark subjects with known user privacy-related bugs, we are unable to evaluate our technique on such benchmarks.

**Functional testing of mobile apps**. Our work focuses on validating the correctness of user privacy functionalities of mobile apps. However, most of existing automated testing techniques (*e.g.*, [14, 26, 38, 40, 59, 73]) in this field are limited to finding crashing bugs. Because they do not accept functional specifications but use runtime exceptions as the oracle [80]. Thus, they *cannot* find the user privacy-related functional bugs like PDTDroid. To find functional bugs, some work (*e.g.*, Genie [60], SetDroid [63, 64], DLD [51], Thor [1]) uses metamorphic testing to generate automated oracles. However, it is difficult to derive the metamorphic relations to capture different properties of user privacy related functionalities. DiffDroid [17] and SPAG-C [35] compare the GUI pages between two different devices (*i.e.*, differential testing) to find device-specific compatibility issues. However, user privacy-related functional bugs are not likely device-specific. Thus, they do not work in our setting. Odin [72] compares the app behaviors after appending the same events to the test inputs terminating at similar GUI layouts. However, it cannot check specific specifications or measure the degree of inspection, so it *cannot* be used in our test scenario (pre-release inspection of industrial app's new version).

**Property-based testing.** Property-based testing (PBT) was first formally introduced by Fink *et al.* [19] and popularized by Claessen and Hughes [11]. There are some work applying PBT in mobile apps. Liam *et al.* propose Quickstrom, which uses LTL to specify the behaviors of web apps as temporal properties, and automatically generates tests for property checking. However, Quickstrom requires users to provide LTL formulas as input, which describes the sequence of GUI pages rather than the temporal logic of complex app states. These required LTL formulas *cannot* characterize privacy specifications involving multiple users. Lam *et al.* [30] design *ChimpCheck* that applies property-based testing in Android apps. However, ChimpCheck uses the *assertions* in the user-provided example-based tests for testing a single function. Thus, the privacy-related bugs involving relationships between multiple functions *cannot* be detected by ChimpCheck (*e.g.*, following users and searching users are the two independent functions). Sun *et al.* [62] introduce a property-based fuzzing approach to detect data manipulation errors in Android apps. Their approach is based on the model-based properties specified by human for testing. Due to the inability to characterize temporal logic, this property *cannot* be used to detect the bugs we are targeting. Moreover, PBT is also applied in other testing scenarios, *e.g.*, RESTful APIs [29], Scratch programs [58], Java Programs [47], OCaml programs [46], Ros programs [52], and Coq programs [31], which cannot be applied for Android apps.

**Model checking and runtime verification.** Model checking [15] and runtime verification [7] are also commonly used to detect security issues that violate specific specifications in apps. Some work applies model checking to mobile apps [6, 70]. However, for large-scale industrial apps, building their models manually is difficult. Although there are some automated modeling techniques [4, 34], they can only obtain models that describe UI transitions, and *cannot* characterize the privacy-related properties of apps. Some work applies runtime verification techniques to mobile apps [55, 61]. Conceptually, our approach differs from runtime verification. Runtime verification checks whether a single execution of the system violates the given specifications by inserting stakes in the system. However, our approach observes the state of the system externally and uses the specification to guide the construction of test cases to find violations. In addition, our approach does not require access to the app source code, which is thus more flexible and accessible.

**Translating natural language texts to LTL formulas.** As far as we know, there are few comprehensive solutions that can translate unrestricted, natural English texts into general LTL formulas [8, 12]. Over the years, some researchers have developed the techniques for translating natural language texts into LTL formulas in specific domains, *e.g.*, aerospace [43, 53], robot [24, 48, 71] and other specific systems [21, 81]. However, these domain-specific methods are difficult to apply to our scenario directly. To our knowledge, there is no English-to-LTL technique for Android apps. To this end, we employ lightweight natural language processing techniques and establish transformation rules to accomplish the target conversion.

## 6 CONCLUSION

In this paper, we introduce a property-driven testing approach to validate privacy functionalities against given privacy specifications. Each privacy specification is expressed as an LTL formula and converted into an automaton. This automaton will enable automatic detection of specification violations in the app and guide the GUI testing. We implemented our approach as a tool, PDTDroid, which can efficiently and sufficiently check given specifications and report any inconsistent app behaviors. The evaluation of PDTDroid on seven popular social media apps demonstrates the effectiveness and practicality of our proposed approach. In our experiment, PDTDroid found 22 previously unknown inconsistency issues.

## ACKNOWLEDGMENTS

We thank the anonymous FSE reviewers for their valuable feedback and the members (Ping Yang, Zhao Zhang, Chao Peng, Jian Yan) from ByteDance's Software Engineering and Security lab for the discussions/feedback on our work. This work was partially supported by National Key Research and Development Program (Grant 2022YFB3104002, 2020AAA0107800), NSFC Project (No.62072178), Shanghai Collaborative Innovation Center of Trusted Industry Internet Software, ByteDance Research Fund, "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (No.22510750100), the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004), and Grant SCITLAB-30009 of Intelligent Terminal Key Laboratory of SiChuan Province.

## REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of android test suites in adverse conditions. In *2015 International Symposium on Software Testing and Analysis (ISSTA)*. 83–93.

[2] Carina Andersson and Per Runeson. 2002. Verification and validation in industry-a qualitative survey on the state of practice. In *Proceedings international symposium on empirical software engineering*. 37–47.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.

[5] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering* 41, 9 (2015), 866–886.

[6] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. 2017. Towards model checking android applications. *IEEE Transactions on Software Engineering* 44, 6 (2017), 595–612.

[7] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to runtime verification. In *Lectures on Runtime Verification*. 1–33.

[8] Andrea Brunello, Angelo Montanari, and Mark Reynolds. 2019. Synthesis of LTL formulas from natural language texts: State of the art and research directions. In *26th International symposium on temporal representation and reasoning (TIME)*.

[9] Duc Bui, Yuan Yao, Kang G Shin, Jong-Min Choi, and Junbum Shin. 2021. Consistency analysis of data-usage purposes in mobile apps. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2824–2843.

[10] ByteDance. 2023. TikTok. Retrieved 2024-2 from https://www.tiktok.com/.

[11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *fifth ACM SIGPLAN international conference on Functional programming (ICFP)*. 268–279.

[12] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. *arXiv preprint arXiv:2303.04864* (2023).

[13] Louis DeNicola. 2023. *How to Manage Your Privacy Settings on Social Media*. Retrieved 2024-2 from https://www.experian.com/blogs/ask-experian/how-to-manage-your-privacy-settings-on-social-media/.

[14] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *42nd International Conference on Software Engineering (ICSE)*. 1–12.

[15] E Allen Emerson and Edmund M Clarke. 1980. Characterizing correctness properties of parallel programs using fixpoints. In *International Colloquium on Automata, Languages, and Programming*. 169–181.

[16] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[17] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 308–318.

[18] Lauren Feiner. 2018. Facebook's worst year ever is now over. Here's how its scandals affected the stock. Retrieved 2024-2 from https://www.cnbc.com/2018/12/31/how-facebooks-stocked-fared-through-privacy-scandals-in-2018.html.

[19] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (1997), 74–80.

[20] Paul Gastin and Denis Oddoux. 2001. Fast LTL to Büchi Automata Translation. *computer aided verification* (2001).

[21] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. 2014. ARSENAL: Automatic Requirements Specification Extraction from Natural Language. *arXiv: Computation and Language* (2014).

[22] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing: 5th International Conference (TRUST)*. 291–307.

[23] Google. 2024. *Google Play*. Retrieved 2024-2 from https://play.google.com/store.

[24] Nakul Gopalan, Dilip Arumugam, Lawson Wong, and Stefanie Tellex. 2018. Sequence-to-sequence language grounding of non-Markovian task specifications. In *Robotics: Science and Systems XIV*.

[25] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th international conference on software engineering*. 1025–1035.

[26] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 269–280.

[27] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G Shin, and Karl Aberer. 2018. Polisis: Automated analysis and presentation of privacy policies using deep learning. In *27th USENIX Security Symposium (USENIX Security 18)*. 531–548.

[28] Syed Ibrahim Imtiaz, Saif ur Rehman, Abdul Rehman Javed, Zunera Jalil, Xuan Liu, and Waleed S Alnumay. 2021. DeepAMD: Detection and identification of Android malware using high-efficient Deep Artificial Neural Network. *Future Generation computer systems* 115 (2021), 844–856.

[29] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 131–141.

[30] Edmund SL Lam, Peilun Zhang, and Bor-Yuh Evan Chang. 2017. ChimpCheck: property-based randomized test generation for interactive apps. In *2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 58–77.

[31] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang.* OOPSLA (2019), 181:1–181:29.

[32] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, Vol. 1. 280–291.

[33] Shuai Li, Zhemin Yang, Yunteng Yang, Dingyi Liu, and Min Yang. 2024. Identifying Cross-User Privacy Leakage in Mobile Mini-Apps at A Large Scale. *IEEE Transactions on Information Forensics and Security* (2024).

[34] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[35] Ying-Dar Lin, José F. Rojas, Edward T.-H. Chu, and Yuan-Cheng Lai. 2014. On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. *IEEE Trans. Software Eng.* (2014), 957–970.

[36] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[37] Christopher Mann and Artem Starostin. 2012. A framework for static detection of privacy leaks in android applications. In *27th annual ACM symposium on applied computing*. 1457–1462.

[38] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *25th International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

[39] Daniel D McCracken and Edwin D Reilly. 2003. Backus-naur form (bnf). In *Encyclopedia of Computer Science*. 129–131.

[40] Monkey Team. 2023. *Android Monkey*. Retrieved 2024-2 from https://developer.android.com/studio/test/monkey.

[41] Madhavan Mukund. 1997. Linear-time temporal logic and Büchi automata. *Tutorial talk, Winter School on Logic and Computer Science, Indian Statistical Institute, Calcutta* (1997), 8.

[42] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. 2011. Natural language processing: an introduction. *Journal of the American Medical Informatics Association* 18, 5 (2011), 544–551.

[43] Allen P. Nikora and Galen Balcom. 2009. Automated Identification of LTL Patterns in Natural Language Requirements. *International Symposium on Software Reliability Engineering* (2009).

[44] Yuuki Nishimoto, Naoya Kajiwara, Shinichi Matsumoto, Yoshiaki Hori, and Kouichi Sakurai. 2013. Detection of Android API Call Using Logging Mechanism within Android Framework. *Lecture notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* (2013).

[45] Business of Apps. 2024. Digital 2024: Global Overview Report. Retrieved 2024-2 from https://datareportal.com/reports/digital-2024-global-overview-report.

[46] Sumit Padhiyar and KC Sivaramakrishnan. 2021. ConFuzz: Coverage-guided property fuzzing for event-driven programs. In *Practical Aspects of Declarative Languages: 23rd International Symposium (PADL)*. 127–144.

[47] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 398–401.

[48] Roma Patel, Ellie Pavlick, and Stefanie Tellex. 2020. Grounding Language to Non-Markovian Tasks with No Supervision of Task Specifications.. In *Robotics: Science and Systems*.

[49] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 46–57.

[50] Junit quickcheck Team. 2024. Junit-quickcheck. Retrieved 2024-2 from https://github.com/pholser/junit-quickcheck.

[51] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 141–152.

[52] André Santos, Alcino Cunha, and Nuno Macedo. 2018. Property-based testing for the robot operating system. In *9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 56–62.

[53] Tainã Santos, Gustavo Carvalho, and Augusto Sampaio. 2018. *Formal Modelling of Environment Restrictions from Natural-Language Requirements*.

[54] Christian Schindler, Müslüm Atas, Thomas Strametz, Johannes Feiner, and Reinhard Hofer. 2022. Privacy leak identification in third-party Android libraries. In *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*. 1–6.

[55] Bradley Schmerl, Jeffrey Gennari, Javier Cámara, and David Garlan. 2016. Raindroid: A system for run-time mitigation of Android intent vulnerabilities [poster]. In *Symposium and Bootcamp on the Science of Security*. 115–117.

[56] Kavita Sharma and Brij B Gupta. 2019. Towards privacy risk analysis in android applications using machine learning approaches. *International Journal of E-Services and Mobile Applications (IJESMA)* 11, 2 (2019), 1–21.

[57] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in android application code. In *38th International Conference on Software Engineering*. 25–36.

[58] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 165–175.

[59] 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 *2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*. 245–256.

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

[61] Haiyang Sun, Andrea Rosa, Omar Javed, and Walter Binder. 2017. ADRENALIN-RV: Android runtime verification using load-time weaving. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 532–539.

[62] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1088–1100.

[63] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 204–215.

[64] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. 2023. Characterizing and finding system setting-related defects in android apps. *IEEE Transactions on Software Engineering* (2023).

[65] Hypothesis Team. 2023. Hypothesis. Retrieved 2024-2 from https://github.com/HypothesisWorks/hypothesis.

[66] Quickcheck Team. 2023. Quickcheck. Retrieved 2024-2 from https://github.com/BurntSushi/quickcheck.

[67] Spot Team. 2023. Spot. Retrieved 2024-2 from https://spot.lre.epita.fr/.

[68] Stanza Team. 2023. Stanza. Retrieved 2024-2 from https://stanfordnlp.github.io/stanza/.

[69] uiautomator2 Team. 2023. *uiautomator2*. Retrieved 2024-2 from https://github.com/openatx/uiautomator2.

[70] Heila Van Der Merwe, Brink Van Der Merwe, and Willem Visser. 2012. Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[71] Christopher Wang, Candace Ross, Yen-Ling Kuo, Boris Katz, and Andrei Barbu. 2020. Learning a natural-language to LTL executable semantic parser for grounded robotics. *arXiv: Computation and Language,arXiv: Computation and Language* (2020).

[72] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 434–446.

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

[74] Yin Wang, Ming Fan, Junfeng Liu, Junjie Tao, Wuxia Jin, Qi Xiong, Yuhao Liu, Qinghua Zheng, and Ting Liu. 2023. Do as You Say: Consistency Detection of Data Practice in Program Code and Privacy Policy in Mini-App. *arXiv preprint arXiv:2302.13860* (2023).

[75] WEditor Team. 2023. *WEditor*. Retrieved 2024-2 from https://pypi.org/project/weditor/.

[76] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

[77] James A Whittaker. 2009. *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. Pearson Education.

[78] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. *Information Security* (2012).

[79] Xiaomi. 2024. *GetApps*. Retrieved 2024-2 from https://global.app.mi.com/.

[80] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1319–1331.

[81] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. 2015. Formal consistency checking over specifications in natural languages. *Design, Automation, and Test in Europe,Design, Automation, and Test in Europe* (2015).

[82] Le Yu, Xiapu Luo, Jiachi Chen, Hao Zhou, Tao Zhang, Henry Chang, and Hareton KN Leung. 2018. Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies. *IEEE Transactions on Software Engineering* 47, 2 (2018), 221–242.

[83] Win Zaw Zarni Aung. 2013. Permission-based android malware detection. *International Journal of Scientific & Technology Research* 2, 3 (2013), 228–234.


Received 2024-02-08; accepted 2024-04-18