MoSSOT: An Automated Blackbox Tester for Single Sign-On Vulnerabilities in Mobile Applications

Shangcheng Shi
The Chinese University of Hong Kong
ss016@ie.cuhk.edu.hk

Xianbo Wang
The Chinese University of Hong Kong
wx017@ie.cuhk.edu.hk

Wing Cheong Lau
The Chinese University of Hong Kong
wclau@ie.cuhk.edu.hk

ABSTRACT

Mobile applications today increasingly integrate Single Sign-On (SSO) into their account management mechanisms. Unfortunately, the involved multi-party protocol, i.e., OAuth 2.0, was originally designed to serve websites for authorization purpose. Due to the complexity of the adapted protocol, a large number of insecure SSO implementations still exist in the wild. Although the security testing for real-world SSO deployments has attracted considerable attention in recent years , existing work either focuses on websites or relies on the manual discovery of specific and previously-known vulnerabilities. In the paper, we design and implement MoSSOT (Mobile SSO Tester), an automated blackbox security testing tool for Android applications utilizing the SSO services from three mainstream service providers. The tool detects the vulnerabilities within the practical SSO implementations by fuzzing related network messages. We used MoSSOT to examine over 500 first-tier third-party Android applications from US and Chinese app markets. According to the test result, around 72% of the tested applications incorrectly implement SSO and are thus vulnerable. Besides, our test identifies an unknown vulnerability as well as a new variant, in addition to four known ones. The vulnerabilities enable the attacker to illegally log into the mobile applications as the victims or gain access to the protected resources. MoSSOT has been released as an open-source project.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

OAuth 2.0; Single Sign-On; Security Testing; Mobile App Authentication

ACM Reference Format:
Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau. 2019. MoSSOT: An Automated Blackbox Tester for Single Sign-On Vulnerabilities in Mobile Applications. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3321705.3329801

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329801

1 INTRODUCTION

As the open standard for authorization, OAuth 2.0[1] enables end-users to grant third-party applications the data access right to their private resources stored on the service provider. Driven by the broad adoption of OAuth and the boom of mobile apps, many prestigious Identity Providers (IdPs), *e.g.*, Facebook and Sina Weibo, have recently tailored OAuth to support SSO for third-party mobile apps, which act as Relying Party (RP) under the context of OAuth. The IdPs also provide RPs SDKs to integrate their SSO services and documents to follow. Therefore, a mobile RP app, *e.g.*, Ctrip, can authenticate a user based on his/ her profile from the IdP, *e.g.*, Sina Weibo, without requiring other identity credentials, *e.g.*, password.

It is worth to note that OAuth was initially designed to provide secure authorization service for web applications. The protocol is actually re-purposed for authentication when used for SSO in mobile platforms. However, few specifications can explicitly guide developers to authenticate users across platforms. Considering the complexity of multi-party authentication and authorization in OAuth, average developers are prone to misinterpret the protocol and do not know how to properly deploy the mobile SSO services. As a result, various vulnerabilities, *e.g.*, Profile Vulnerability [49] and App Secret Disclosure [11, 47], have been discovered by the literature in recent years. Although the SSO security testing has received increasing attention [47, 50, 53], all the existing studies either work on web applications or rely on the manual or semi-automatic discovery of known and specific vulnerabilities.

Nevertheless, it is not trivial to develop the security testing tool for mobile SSO due to the following three challenges.

- *Challenge 1: Difficult to Manipulate App State*
  The mobile apps are stateful, which may involve some client-side logic, *e.g.*, encoding the user credential, so it is hard to manipulate the expected app state for the testing.
- *Challenge 2: Heterogeneous SSO Customizations*
  Given the platform difference, both IdPs and RPs tend to customize the SSO service and the tool needs to cater to the app-specific implementations adaptively.
- *Challenge 3: Unexpected App Behaviors*
  The mobile app may perform unexpectedly, *e.g.*, loading dynamic pop-ups, during execution. Thus, detecting the abnormal state and recovering the app from it are difficult.

To tackle the challenges, we design a model-based blackbox security testing framework and further implement it into MoSSOT (already open sourced at [34]). Though we may manually extract the SSO-related logic from the app via reverse-engineering to manipulate the app state, it is not scalable to perform the analysis on

[1]For the rest of the paper, we use OAuth to denote OAuth 2.0 and other OAuth 2.0-based protocols, *e.g.*, OpenID Connect (OIDC), if not specified otherwise.

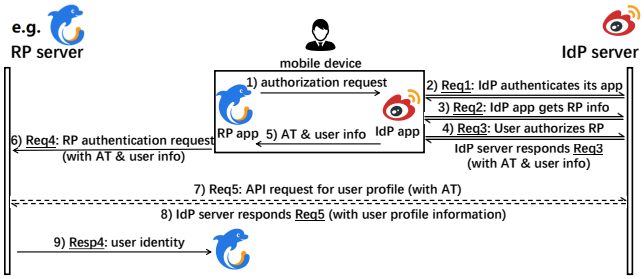

Figure 1: Implicit flow of OAuth 2.0 for mobile platforms

each app under the blackbox setting. As a workaround, the tool performs an SSO in each test such that the app will reach the expected state halfway. Thus, we develop a module to automatically simulate related UI operations. Then, the tool can complete normal SSOs and learn the customizations from the resultant network traffic to generate meaningful test cases. Finally, we build a robust testing architecture with the capability of real-time app state tracking and exception recovery. We summarize our contributions as follows:

- We propose a blackbox security testing framework for the SSO implementations in Android apps.
- We design and implement the proposed framework into an automatic testing tool, MoSSOT.
- Using MoSSOT, we conduct security assessments for over 500 top-ranked RP apps. Our tool has identified a previously-unknown vulnerability, in addition to four known ones, as well as a new variant.

The rest of the paper is organized as follows. Section 2 introduces the adapted OAuth protocol flows for mobile apps. Section 3 describes the system architecture of MoSSOT and more design details are given in Section 4. Section 5 summarizes the testing result as well as the detected vulnerabilities. Section 6 discusses some limitations of MoSSOT and presents potential solutions. We review related work in Section 7 and conclude the paper in Section 8.

2 BACKGROUND

The OAuth framework consists of three parties: IdP, RP, and User (User-Agent). Under a mobile environment, IdP and RP map to the backend servers of the IdP (IdP server) and the third-party mobile app (RP server) respectively. Meanwhile, the User-Agent switches to the mobile apps of the IdP (IdP app) and RP (RP app). For ease of presentation, we use the notations in the parentheses to denote the four parties in the rest of the paper.

The target of OAuth, when used for SSO in mobile apps, is for the IdP server to issue a credential, *e.g.*, access token (AT) for OAuth 2.0, to the RP server. Then, the RP server can use the credential to extract user profiles from the IdP server for authentication and finally logs the user in. Among the four types of authorization flows defined in OAuth 2.0 [21], implicit flow and authorization code flow are adopted by the mobile platform. Within the studied IdPs, Sina Weibo adopts the implicit flow, while WeChat utilizes the latter.

OpenID Connect [37], on the other hand, builds an identity layer on the top of OAuth for more efficient authentication, as OAuth was originally designed for authorization. Facebook adopts OpenID Connect in their SSO service. We introduce call flows of the two authorization flows within OAuth and OpenID Connect here.

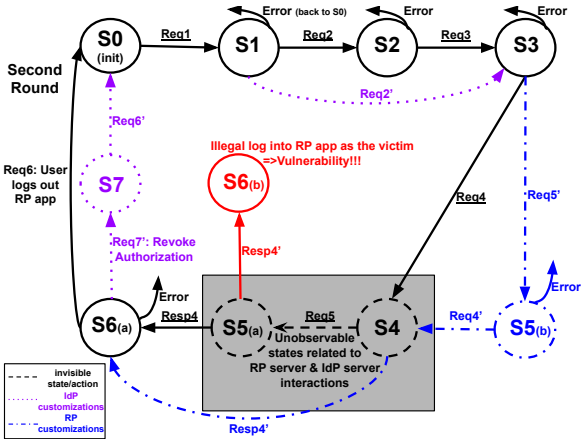

Figure 2: One state machine example for Sina Weibo

- The path in black represents the normal SSO process in Fig. 1.
- The colored dashed part stands for IdP/ RP customizations.

2.1 The Implicit Flow of OAuth 2.0

Since OAuth was not originally designed for mobile apps, neither the RFC nor the IdPs provide a complete call-graph diagram to mobile RP developers. After revising the specifications [21, 40], one practical realization of the implicit flow is shown in Fig. 1.

In the implicit flow, the access token (AT in Fig. 1) goes through the mobile device (Step 4 to 6). Then, the access token is consumed by the RP server to extract user profiles from the IdP server in Step 7, so it must be protected by Transport Layer Secure (TLS). Besides, the RP server should authenticate the user based on the user profile information returned in Step 8 after verifying the access token.

2.2 The Authorization Code Flow of OAuth 2.0

The authorization code flow of OAuth 2.0 (Fig. 7 in Appendix. A.1) is actually an augmented implicit flow. Compared to the implicit flow (in Fig. 1), the IdP server responds an intermediate token, *i.e.*, code, instead to the IdP app. After receiving the code, the RP server needs to exchange it for an access token (Step 6 to 7 in Fig. 7).

There are two important properties associated with the authorization code flow: (1) the code alone is useless because the IdP server needs to first verify the appended app_secret (Step 6 in Fig. 7), a pre-shared secret as the identity proof of RP to IdP, before issuing the access token. (2) the code *must* be short-lived and single-use.

Due to the properties, the authorization code flow is more secure than the implicit flow at the cost of one more round-trip between servers. Besides, RP needs to maintain the app_secret on its server.

2.3 The OpenID Connect Protocol

The OAuth 2.0 suffers high-latency when adapted for SSO, *e.g.* Step 6 to 9 in Fig. 7. To overcome the limitation, some IdPs, *e.g.*, Facebook, have adopted OpenID Connect [37] (OIDC) for their SSO services. OIDC builds an identity layer upon the OAuth 2.0. OIDC supports three types of authentication flow, but only implicit flow is utilized in mobile platforms, so we only concentrate on it in the paper.

As depicted in Fig. 8, the major extension that OIDC makes to OAuth is a newly-introduced token called id token dedicated for authentication. The id token is in the form of JSON Web Token [26] and digitally signed by the IdP server. In particular, it contains

a user identifier (1234 in Fig. 8) and can be extracted by the RP server. Given that the signature cannot be tampered or forged by the attacker, the RP server can identify the user directly based on the identifier without the extra communication with the IdP server.

2.4 The Customized Implementations by IdPs

The diagrams in the section only describe the basic scenario, while IdPs tend to add the following customizations to their SSO service.

Adoption of WebView. When the IdP app is not installed, its role will be replaced by an embedded web browser in the RP app.

Authorization Revocation. Although how to revoke the authorization is not defined in the protocol specifications, some IdPs, *e.g.*, Facebook, provide an app management page on which the user can review and revoke all the authorized RPs.

Automatic Authorization. When a user has authorized an RP app before, the IdP may skip the user-consent for authorization, *e.g.*, Step 3 in Fig. 1. Thus, the login process involves no user interactions.

2.5 Threat Model

In our threat model, the target of an attacker is to break the authentication of a mobile app, *i.e.*, logging into an RP app illegally with the identity not belonging to himself. We assume the mobile device is not compromised, the IdP is benign and the communication between the IdP server and RP server is well-protected.

In particular, we consider two types of attackers, namely (1) a network attacker and (2) a malicious RP attacker. A network attacker can intercept, replay or tamper the unencrypted network traffic through the victim's device, while a malicious RP attacker may act as a benign RP to steal the credentials of the victim.

The attacker is also able to log into benign RPs as normal users and analyze or modify the network traffic of his own. Besides, the attacker may decompile the IdP/ RP app.

3 SYSTEM OVERVIEW

In the section, we introduce the system architecture of MoSSOT, followed by the detailed workflow.

3.1 Overall System Architecture

To traverse every possible path in the call flow of OAuth (*e.g.*, Fig. 1), we build our tool on the top of PyModel [23], an open-source model-based testing (MBT) framework. PyModel takes a state-machine-based system model as input and automatically generates test cases, which can formally enumerate all the possible paths in the given model and thus guarantee test coverage to some extent.

Fig. 3 presents the system architecture of MoSSOT. The tool is composed of five modules: UI Explorer, Test Engine, Test Learner, System Model, and Test Oracle. The framework tackles the challenges discussed in Section 1 and can be divided into three portions.

- UI Explorer (Section 4.1), is responsible for solving *Challenge 1*. The module automatically explores the UI widgets within the mobile apps that need triggering to reach the desired destination, *e.g.*, login page. Then, Test Engine (Section 4.3.1) can automatically perform SSOs and drive the app to the expected state for the actual testing.
- Test Learner and System Model (Section 4.2) cope with *Challenge 2*. We first construct an initial model manually based on protocol specification [21, 37] and IdP documentation (*e.g.*, [14]), which caters to the IdP customizations (mentioned in Section 2.4). Then, Test Learner analyzes the network traffic from normal SSOs to learn the app-specific implementations by RPs, which complement the initial model.
- Test Engine and Test Oracle (Section 4.3) tackle *Challenge 3*. To execute concrete test cases, Test Engine performs SSOs and drives the mobile app to the expected state. At the same time, Test Engine feeds back observations to Test Oracle for monitoring the state change and identifies potential vulnerabilities. Besides, once unexpected app behaviors are detected, the tool will try to recover the app to the correct state.

We elaborate on the design of each portion in the next section.

3.2 Workflow of MoSSOT

Given the APK of the RP app, MoSSOT works as follows:

(1) The input APK is unpacked to extract necessary information, *e.g.*, package name. Then, the app is installed and its information is passed to UI Explorer and Test Learner.
(2) UI Explorer executes the RP app and tries to find the set of widgets that lead to a certain page, *e.g.*, SSO login page. The result is output to Test Engine for automatic UI driving.
(3) Test Learner requests Test Engine to perform normal SSOs on the RP app. The network traces between the device and the RP or IdP server are captured.
(4) Test Learner analyzes the network traces and learns the app-specific customizations.
(5) Based on the specifications and documents (*e.g.*, [21] and [14]), we build an initial model beforehand, which is then augmented with the learned app-specific customizations.
(6) With System Model, MoSSOT automatically generates test cases in the form of abstract HTTP(S) requests. Then, Test Engine performs SSOs and tampers the network traffic according to the test cases.
(7) Test Oracle collects (UI & network) observations from Test Engine. If the app is detected to perform unexpectedly, the module will try to recover it from the error state.
(8) Test Oracle extracts the expected system behavior from System Model and compares it with the observations to determine whether the latter is normal or not.
(9) For any abnormal behavior, MoSSOT determines whether it is exploitable or not. If so, the tool will output the test case.
(10) If the test case leads to the expected result, *i.e.*, logging into the RP with the identity in the IdP, System Model will remove associated test cases and then output the next one to Test Engine (*i.e.*, Step 6). The system continues the iterative process until achieving the pre-defined requirements.

4 DETAILED DESIGN OF MOSSOT

In the section, we first introduce the framework of UI automation. Then, we describe how to learn the customizations within practical SSO implementations for constructing a comprehensive system model. Finally, we talk about the actual model-based testing, give a running example, and discuss some implementation challenges.

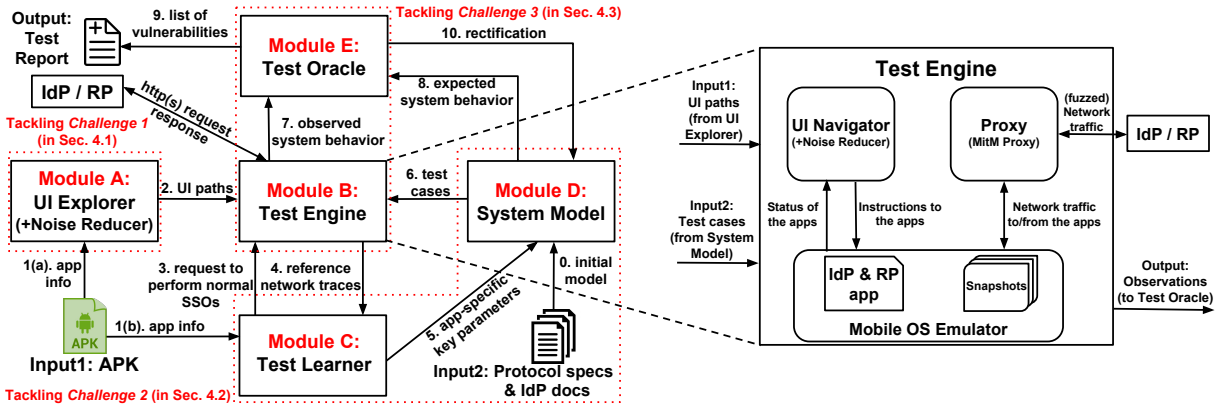

Figure 3: System architecture of MoSSOT

4.1 UI Automation Framework

Our testing system requires automated and repeated state manipulation on a large number of mobile apps with only blackbox level information as described in *Challenge 1*. To guarantee scalability, efficiency, and robustness, we have to complete the following tasks:

(1) *Targeted UI search*: To support large-scale testing, our testing system needs to automatically find the path to the target login page of an RP app, where efficiency and accuracy are the major requirements.
(2) *Robust recording and replaying*: Our testing needs to perform the login process repeatedly so a convenient UI path recording method and a robust replay tool are essential.
(3) *Handling random UI components*: Many apps tend to dynamically load remote resources and display them, *e.g.*, advertisements, which introduces randomness and makes the UI automation complex and unstable.

We design three modules, namely UI Explorer, UI Navigator, and Noise Reducer, to complete the three tasks. UI Explorer (Module A in Fig. 3) is used to search for the set of UI elements that drive the app to the destination page. The result is saved as the UI path and later fed into UI Navigator, which is integrated into Test Engine (Module B in Fig. 3) for replaying the UI paths to perform SSOs. On the other hand, Noise Reducer provides a consistent environment so that the other two can operate on the tested app robustly. We discuss each of the three modules in detail here.

4.1.1 *Modeling the UI System*

Before the discussion, we need to first clarify a few terminologies. (1) *element*: Android UI layout, is made up of UI widgets with several attributes. However, there is no single attribute that can act as a unique identifier. In our framework, we define an *element* as an object with the following properties: *type, text, id, desc, clickable, Xpath, screenshot*. Some of these properties directly map to XML attributes in the UI layout, while the others store additional information, *e.g.*, *screenshot*. The identifier of an *element*, either a single property or a combination of some, is generated on the fly to guarantee its uniqueness. (2) *page*: In most time, *pages* directly map to Android activities. However, the content and layout may change dramatically under the same activity, which appears to users as multiple *pages*. Our abstracted *page* object hence is identified by both activity name and page layout. (3) *UI path*: A *UI path* is defined as an ordered list of *elements* that need triggering and a set of matching conditions to identify the target *page*.

4.1.2 *UI Explorer*

UI Explorer (Module A in Fig. 3), as indicated by its name, automatically searches for the *UI path* towards a destination. The input of the module is simply a desired destination. In our SSO testing, the destination is configured to be SSO login *page* and we developed two exploration algorithms. Algorithm I is a novel heuristic-based algorithm called level-based keyword scan (LKS), designed for efficiency, while Algorithm II is a depth-first-search (DFS) algorithm with custom prioritizing, built for higher accuracy. Details of the algorithms are given in the following paragraphs. For scalability, both algorithms purely rely on UI information instead of code analysis like done in [4, 7]. Our approach guarantees that obfuscated or packed apps can also be tested. Besides, unlike previous work [2, 12, 20, 32] focusing on test coverage, we aim at finding the target UI *element*/*page* with higher accuracy and speed.

Algorithm I (LKS). The idea of the algorithm is based on the observation that there are some general semantic patterns when navigating to the SSO login page. For instance, one commonly seen path is *home* → *profile* → *login*. To translate it into a heuristic algorithm, we start with crafting a 2D-list containing keywords commonly appear in each stage of *pages* with semantically similar keywords in the same inner list (level). Based on that, the algorithm first tries to find a matching keyword in the level representing the stage of *page* closest to the destination, then tries consecutive levels till the farthest one. Given the space restriction, we elaborate on the algorithm and the generation of keyword list in Appendix B.1.

Algorithm II (DFS). The algorithm is an adapted DFS, where we model the connection of *elements* and *pages* as a directed graph. In the graph, *pages* and *elements* are vertices and transitions between *pages* (via interacting with *elements*) is modeled as directed edges. Notice that the graph could be cyclic, so our adapted DFS needs to detect cycles and cut the loops. Meanwhile, within each *page*, there are normally tens of *elements*. Instead of random searching, a smart ordering improves efficiency greatly. We achieve this by assigning weights to edges and do a prioritized DFS. In addition, in our case, both the weight and search depth should be bounded as most of the low-weight *elements* are not worth trying and login *page* normally sits at a shallow path. Lastly, limited by Android UI

behavior, we cannot always simply jump back to the parent vertex, so we achieve that by returning to the root and revisit the path. More details of the algorithm are given in Appendix B.2.

4.1.3 *UI Navigator*

The module is for *UI path* replay and integrated into Test Engine as shown in Figure 3. [27] shows that all state-of-the-art record and replay tools have limitations, which make them less suitable for production. Our tool, instead of trying to solve this general problem, targets at providing a robust and efficient UI replay with only simple user interactions. Instead of using coordinate-driven navigation as done in [35], this module aims to find and trigger widgets by their attributes, *e.g.*, text, id, and even XPath, in a way similar to [1]. This reason behind is that the former is sensitive to noise, *e.g.*, misaligned page, and scrolling. The exclusive feature of our tool is the dynamic *UI path* handling. Given a *UI path* recorded in JSON format, the module visits *elements* one by one till all of them are consumed and the destination is matched. During the process, it can happen that some target *elements* cannot be found in the current *page*. Then, Noise Reducer will be triggered to detect and clear noise until UI Navigator can resume the *UI path* replay.

4.1.4 *Noise Reducer*

The randomness of UI components motivates us to design Noise Reducer, which can heuristically handle the contents considered as noises, *e.g.*, popups, advertisements, and loading *pages*. The module first classifies the noises based on the features extracted from the current UI layout, *e.g.*, number of clickable *elements*, keywords, and widget size. Then, for different types of noises, corresponding handling strategies will be applied. For instance, the handler will try to bypass a welcome *page* by swiping. Once Noise Reducer starts, it runs in a loop until it cannot detect any noise. In our framework, Noise Reducer is invoked passively when normal actions (*e.g.*, taps) fail in the other two modules.

4.2 Modeling the Mobile SSO Protocols

System Model (Module D in Fig. 3) is actually a finite state machine (FSM). The FSM consists of system states and corresponding actions, *i.e.*, the necessary triggers for the next state.

4.2.1 *Constructing the Initial Model*

Firstly, we construct a state machine (*i.e.*, the black path in Fig. 2) based on the normal SSO process, where each action exactly maps to the request/ response in Fig. 1. As shown in Fig. 2, each circle represents a particular state during the SSO process and each edge stands for a certain message (request/ response) leading to a state transition. For example, *S1* represents the state that the IdP app just receives the response of last request (*Req1*), while *Req2* corresponds to the request from the IdP app to its server, seeking RP information.

Nevertheless, as discussed in Section 2.4, IdPs usually customize their SSO services, which is not included in the normal workflow. Thus, we also consider the IdP customizations in the initial model. For example, when the user has authorized the RP before SSO, each IdP performs differently. Among the three IdPs we study, Sina Weibo adopts Automatic Authorization, where *S1* jumps to *S3* in Fig. 2, while Facebook utilizes a different API to process the SSO request. In contrast, WeChat always performs the same as the first

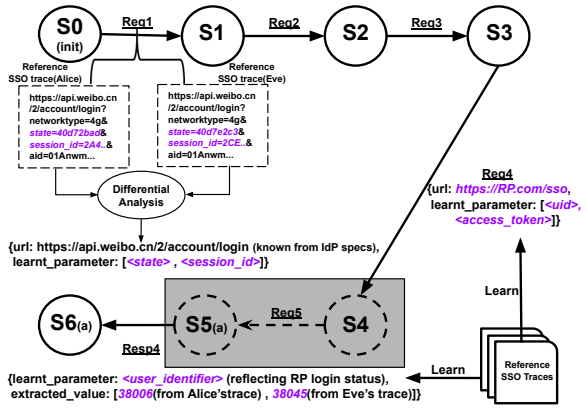

Figure 4: The learning process to augment initial model

- The italics highlighted in purple represent the learnt content.
- To avoid clutter, this example only shows the state transitions during a normal SSO.

SSO attempt. Given the practical scenarios, we separate the involved states into two versions: one for unauthorized login and the other for pre-authorized login. System Model thereby can traverse both situations via *Req7*, which revokes the authorization.

Besides, when analyzing the practical SSO network traffic, we find that some RPs tend to deploy the server-to-server logics, *e.g.*, Step 7 and Step 8 in Fig. 1, on the client side (RP app). Then, the optional interaction between the RP app and IdP server becomes visible to the mobile device. Therefore, we also reserve the corresponding state in our initial model, *i.e.*, *S5(b)* in Fig. 2, which will be adaptively switched on/off according to the actual implementations.

4.2.2 *Learning App-Specific SSO Implementations*

However, the initial state machine is unaware of the app-specific implementations. In other words, key parameters in each action and the realization of *Req4* and *Resp4* (in Fig. 2) are unknown. Thus, Test Learner (Module C in Fig. 3) needs to analyze practical SSO network traffic to learn the implementation details.

Test Learner first requests Test Engine (Module B in Fig. 3) to perform normal SSOs under particular settings, *e.g.*, with different IdP identities (Alice and Eve) and at a different time (logging as Eve again). At the same time, the generated network traffic will be intercepted and saved as reference network traces. As shown in Fig. 4, Test Learner then performs differential analysis on the traces to identify key parameters in each action. To confirm the key parameters, Test Learner replays the request with the parameter removed. If the response remains the same, the parameter is discarded, as the existence of a valuable parameter will affect the consequent response. Ultimately, each action can be represented by URL and learnt key parameters.

Another task of Test Learner is to identify the authentication interaction between the RP app and its server, *i.e.*, *Req4* and *Resp4* in Fig. 4. Unlike the use of OAuth for websites, where the interaction is realized by redirection, the OAuth standard does not define how the RP app should deliver the received credential, *e.g.*, access token, to its backend server. In other words, it is implementation-specific and subject to RP customizations. As such, Test Learner is responsible for learning individual RP customization by examining the reference network traces based on some heuristics. For instance, the edit

distance between the domain name of the request URL and the package name of the tested app (from Step 1(b) in Fig. 3) tends to be small, *e.g.*, douban.com and com.douban.movie. Besides, the response (*Resp4*) should contain a user-identifier that are user-dependent and session-independent. Then, Test Learner will record the user identifier values, *e.g.*, 38006 (Alice) and 38045 (Eve), within the corresponding response (*i.e.*, *Resp4* in Fig. 4), so that Test Oracle (Section. 4.3.2) can identify the RP login status in the test later.

4.2.3 *Generating and Refining Test Cases*

With the learnt parameters in each action, MoSSOT would generate test cases for the actual testing. Given the capability of the attacker (defined in Section 2.5), he is able to tamper SSO-related network traffic. For example, he may replace the access token by a stolen one (from the same or a different RP app) in *Req4* (*i.e.*, Step 6 in Fig. 1). Once the verification within the RP server is incomplete/ incorrect, he may cheat the server into authenticating himself as the victim. Thus, we design four types of test cases based on the learnt key parameters:

(1) Remove or randomize a single parameter;
(2) Replace a single parameter with the one from a parallel session (on the same app with a different identity);
(3) Replace a single parameter with the one from a different session (on a different app with a different identity);
(4) Replace two parameters simultaneously with the ones from a parallel/ different session.

The values for replacement are from the reference network traces, so we will only log into the app with testing accounts without affecting normal users. The property of protocol-defined parameters will also be considered. For example, we prepare one extra unused authorization code for replacement, as it is single-use (Section. 2.2).

The test cases are prioritized in the same order, where a single parameter is fuzzed first. Once the current test case is identified to be redundant, *i.e.*, fuzzing does not affect RP login status, following associated test cases will be ignored (*i.e.*, Step 10 in Fig. 3).

Under the current implementation, we only consider the simultaneous replacement of up to 2 parameters. In principle, MoSSOT can support more complicated cases like the combination of more parameters and different operations, *e.g.*, replacing access token and randomizing uid in *Req4*. However, the number of test cases will increase exponentially and thus slow down the testing.

4.3 Building a Robust Testing Architecture

To execute the test cases from System Model, Test Engine drives the mobile app to the expected state. Then, Test Oracle tracks the app state and identifies potential vulnerabilities, based the real-time observations. The two modules construct a robust testing architecture and thus solve *Challenge 3*.

4.3.1 *Execution of Test Cases Under Test Engine*

Test Engine (in Fig. 3) is made up of three components: UI Navigator, Proxy, and Mobile OS Emulator. We first describe their functionality and then discuss their integration for executing test cases.

UI Navigator. As discussed in Section 4.1.2, the module takes the UI paths from UI Explorer as input to drive the apps in the Mobile OS Emulator for simulating user behaviors.

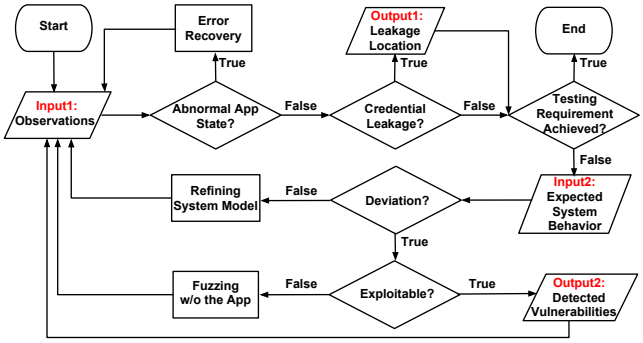

Figure 5: Flow Chart of Test Oracle

Proxy. Since our testing relies on tampering SSO-related network traffic, we set up a *MitMProxy* [33] in Test Engine. As the *MitMProxy* is SSL-enabled, Proxy manages to monitor, intercept or tamper the HTTP(S) traffic from apps to servers.

Mobile OS Emulator. Mobile OS Emulator in Fig. 3 is the execution environment of the IdP app and RP app. MoSSOT supports two types of emulator, *i.e.*, Genymotion [16] and Android Emulator [18]. As the snapshot capability of the latter can recover the app from an error state efficiently, we use it in large-scale testing.

In the actual testing, UI Navigator first sends instructions to the apps (in the emulator) to perform the SSO login process. Meanwhile, UI operations lead the app to the expected state and trigger SSO-related network traffic, which goes through Proxy. Then, Proxy tampers the traffic according to the current test case. Afterward, the observations from Proxy and UI Navigator are output to Test Oracle (Section 4.3.2). Finally, UI Navigator resets the state of the apps to execute the next test case.

4.3.2 *Test Oracle*

Fig. 5 presents the workflow of Test Oracle, which aims to complete the following tasks.

Tracking the App State. Test Oracle keeps monitoring the feedback from UI Navigator to track the app state. Once detecting an abnormal app state, *e.g.*, unexpected crash, the module will try to recover the app to a correct one for resuming the test by (1) resetting and restarting the app or (2) loading a prepared snapshot.

Detecting the Leakage of Credentials. Test Oracle also checks the protection of user/ app identity credentials (*e.g.*, access token and app secret), because the attacker can obtain the credentials via sniffing the SSO sessions of victims/ himself (Section 2.5).

The module takes different strategies to detect the leakage. For the former, it extracts credential values from the interactions between the IdP app and its server, *e.g.*, Step 2 to 4 in Fig. 1, and will generate an alarm once they are sent in plaintext (HTTP). For the latter, we manually collect the list of network APIs provided to the RP server (involving app secret) beforehand and their invocations by the RP app indicate the leakage problem. The leakage locations are then output, *i.e.*, Credential Disclosure and App Secret Disclosure in Table. 2.

Identifying the Vulnerabilities. The module identifies the potential vulnerabilities by comparing the observations with expected behavior (from System Model). If the testing does not lead to deviation, the current test case is classified as redundant and System Model will be rectified accordingly (discussed in Section 4.2.3).

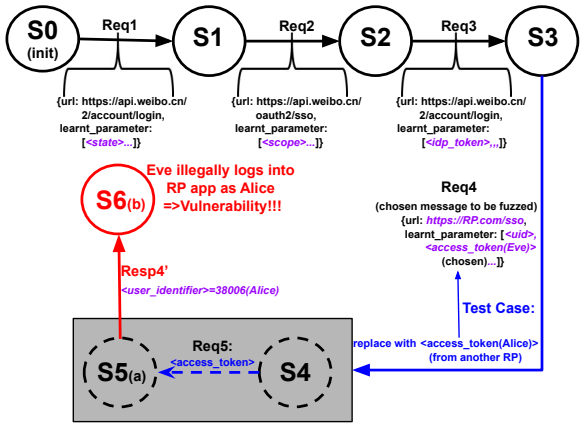

Figure 6: A running example of replacing access token

In contrast, the deviation may not be exploitable. For example, removing access token in *Req4* will cause the error response (Error in Fig. 2). To verify whether the deviation can indeed be exploited, Test Oracle will check RP login status by comparing the user identifier value within the authentication response (*Resp4* in Fig. 2) and the one of Eve from reference network traces (Section 4.2.2). As the tester always logs into the app as Eve during the test, Test Oracle will generate an alarm if the two differ, indicating that the system enters an abnormal state (*S6(b)* in Fig. 2), *i.e.*, logging into the RP app illegally as another user (like Alice).

Fuzzing without the App. Sometimes, the server may detect our fuzzing (*e.g.*, removing access token) and respond with error messages directly. Based on the context, Test Oracle will send tampered requests according to the following test cases directly. Then, Test Oracle can get immediate responses from the server. If error messages disappear, Test Oracle will stop and replay the current test case to investigate its real impact. The functionality helps speeding up the test, as the UI operations are time-consuming.

4.3.3 *Checkpointing and Error Recovery*

Despite the recovery mechanism from Test Oracle, MoSSOT crashes occasionally due to the internal errors in Emulator or UI Navigator. As some modules we use, *e.g.*, Android Emulator [18], are complex systems themselves, we cannot identify the exact causes.

To improve stability, we have implemented a checkpointing function. Once the testing progress remains unchanged for a long time, indicating the anomaly, checkpointing will start to work.

Specifically, the recovery consists of two stages: (1) recover to the interrupted *action* (in System Model) (2) continue the remaining test cases. Once exceptions happen, the checkpointing will resume the interrupted *action* and reset the app state accordingly[2] by loading a prepared snapshot or launching a new emulator instance[3]. Then, MoSSOT continues to execute the unfinished test cases.

4.4 A Running Example of Testing Phase

We give a running example here to illustrate the testing phase, including how MoSSOT executes test cases and detects vulnerability.

As shown in Fig. 6, at a certain moment of our testing, MoSSOT chooses to fuzz access token within *Req4*, where the test case is to replace its value with the one of Alice (from another RP app).

(1) Then, MoSSOT asks Test Engine to perform the login process via SSO and replace the access token within *Req4*, while making no changes to other messages, *e.g.*, *Req1*.
(2) After that, the tampered access token arrives the RP server and is included in the request to the IdP server, *i.e.*, *Req5*.
(3) If the verification on the RP server is incomplete, it will trust the IdP response by mistake, authenticate the tester as Alice, and returns her user identifier (*e.g.*, 38006) to the RP app.
(4) By comparing the recorded user identifier value (from Section. 4.2.2) and the real-time value, Test Oracle identifies that the tester logs into the RP app illegally as Alice, *i.e.*, entering *S6(b)*, indicating the vulnerability within the RP server.

4.5 Additional Implementation Challenges

In the preliminary test, we encountered several practical challenges. We illustrate them here and give the current/ potential solutions.

Certificate Pinning. Both the IdP and RP may apply certificate pinning to protect their mobile apps. In the situation, the app will check the certificate and refuse to work as the certificate belongs to Proxy. Thus, we install a universal unpinning tool on the emulator to bypass the protection within most RP apps.

However, the tool does not work for the Facebook app. As a workaround, we uninstall the app and then WebView (Section 2.4) will be used instead, where certificate pinning is not available. Besides, WebView works almost the same as the Facebook app in SSO except that it utilizes different network APIs.

To generalize our tool, there are two solutions to tackle the certificate pinning. First, we have written a tool to hook HTTP-related functions in Android to tamper our interested data without using Proxy. Second, we may hook the functions within the IdP SDKs to intercept the interactions between the IdP and RP apps. We plan to add the two functionalities into MoSSOT in the future.

API Changes of IdP. During the large-scale testing, we noticed an abnormal number of failure cases for apps with WeChat SSO. After some investigation, we found that WeChat is migrating to a new API with A/B testing. Worse still, in the new API, all the HTTP(S) messages are encoded. Consequently, Test Learner could not extract necessary data from the messages, *e.g.*, code, to identify the interaction between the RP app and its server (Section 4.2.2).

Thus, we develop an XPosed [48] module to hook related functions within the IdP app and force WeChat to use the old API.

Trouble of background animation. During the test of our UI automation framework, we observed the extremely slow reactions when the app contains continues UI animation.

We traced back and discovered a bug in the Android UIAutomator. Later in January 2018, we submitted a patch to Google.

5 EMPIRICAL TESTING

We have implemented MoSSOT in Python with around 12000 lines of code. Using the tool, we managed to assess the practical SSO implementations within 550 RP apps that integrate the service from three major IdPs, namely Sina Weibo, WeChat, and Facebook.

[2]The function may also revoke the authorization to change the server state.
[3]The emulator sometimes is not responsive and we have to start a new one cloned from the template instance.

Table 1: Success rates at different stages (under the fully blackbox & automated setting)

IdP	#Dowloaded Apps	#Screener Output	#UI Explorer Output	#Success Cases
Sina	12872	3322 (25.8%)	767 (23.1%)	196 (25.6%)
WeChat	12872	4692 (36.5%)	822 (17.5%)	226 (27.5%)
Facebook	11064	2095 (18.9%)	436 (20.8%)	128 (29.4%)

5.1 Dataset and Test Setup

We developed crawlers to download 12872 and 11064 Android Apps (in Table. 1) from two third-party Android app store, *i.e.*, Wandoujia [45] and Apkpure [3], respectively in June 2018. As Google Play did not host many Chinese apps with Sina Weibo and WeChat SSO, we used Wandoujia instead. Considering consistency, we chose Apkpure as the dataset source for Facebook apps. According to [41] and [52], they are both highly-ranked third-party app stores.

However, a majority of the apps do not have SSO support. As a preprocessing step, we implement the Screener to heuristically filter out the apps that have no indication of the SSO login integration of the three studied IdPs. The remaining dataset is the initial input to our MoSSOT framework (*i.e.*, Input1 in Fig. 3).

Ultimately, we use MoSSOT to perform the large-scale testing on a machine with a 2.4GHz quad-core CPU and 64GB memory running Ubuntu 16.04. On average, the tool takes 2.85 hours to complete the test of one RP app. Besides, only 550 RP apps completed the test under the fully automated and blackbox setting due to various reasons. We delay the discussion of encountered issues as well as potential solutions to speed up the testing in Section 6.

5.2 Efficiency and Detection Accuracy

Considering efficiency, the test cases are prioritized before execution (as mentioned in Section 4.2.3) so that single test cases (associated with known vulnerabilities) will be executed first. Afterward, the tool turns to execute the combination cases, where two parameters are replaced simultaneously and may lead to the detection of unknown vulnerabilities.

In terms of detection accuracy, there should be no false positives as we confirmed that by manually validating the testing result of 30 randomly-chosen apps, where no false alarms were found. However, there may be false negatives due to following two reasons.

First, there may be a large network delay before receiving the final authentication response, due to server-to-server interactions, *e.g.*, Step 7 and 8 in Fig. 1. We heuristically set a timeout to be 5 seconds and MoSSOT will abort the current test once it is triggered, which can result in false negatives. Second, after fuzzing, the RP server may recognize us as new users and respond with unexpected messages upon the first login, *e.g.*, requirements to enrich user profile, so that Test Oracle may misunderstand the response.

5.3 Security Testing Results

According to the statistics in Table. 2, around 72.4% of the tested apps are susceptible to at least one vulnerability due to their poor implementations on either the client (Android app) or server side.

MoSSOT has detected 4 types of known vulnerabilities manually identified by the previous work, *i.e.*, Access Token Replacement [11, 46], Profile Vulnerability [49], Credential Disclosure [47], and App Secret Disclosure [11, 47]. Moreover, our tool improves the accuracy in detecting App Secret Disclosure by a hybrid method.

In addition to the known vulnerabilities, the tool discovered a new variant of Access Token Replacement and Profile Vulnerability, Augmented Token Replacement, as well as a previously-unknown vulnerability on the IdP side. All the vulnerabilities may be exploited by the attacker to log into the RP as the victim or even impersonate as the benign RP to IdP for conducting privileged operations.

5.3.1 *Discovery of Augmented Token Replacement Attack*

Observations: Table. 2 shows that 41.8% and 51.6% of the tested apps are vulnerable to Access Token Replacement Attack [11, 46] and Profile Vulnerability [49]. The former one replaces the access token from the IdP server, *e.g.*, Step 6 in Fig. 1, and the other tampers the appended user information. Nevertheless, the two vulnerabilities are only feasible when no verifications of the token (access token/ id token) exist on the RP server side, as the user information from its client (*e.g.*, Step 6 in Fig. 1) differs from the one from the IdP server (*e.g.*, Step 8 in Fig. 1). In our test, MoSSOT discovers a variant of the two vulnerabilities that can bypass the checking and affect 57.8% tested apps.

In the original protocol specification [21], the access token is issued on a per-user basis. In contrast, the tokens issued by the three studied IdPs are per-app and per-user based, so RPs should verify the binding between the received access token and itself from the IdP response, *e.g.*, Step 8 in Fig. 1, which tends to be missed. Besides, the access tokens are all bearer tokens [25]. Thus, the attacker can extract the associated user information of victims from the IdP directly with either the stolen (*i.e.*, network attacker) or obtained (*i.e.*, malicious RP attacker) token, *e.g.*, replaying Step 7 in Fig. 1. Consequently, the attacker can inject both the token and its corresponding user information in his own session, *e.g.*, Step 6 of Fig. 1. As a result, the attacker can bypass the aforementioned checking by augmenting the injected access token with related user information and thus cheat the RP.

In terms of exploiting the vulnerability, there are two major differences between WeChat and the others. First, WeChat customizes the operation of extracting user information and requires the corresponding openid (an app-specific user id) in the request (Step 8 in Fig. 7). However, according to our manual test, WeChat server actually will not check its value so the attacker can still extract the user information with solely an access token. Second, WeChat adopts the authorization code flow (Section 2.2) such that the access token is invisible to the attacker-controlled handset in the normal flow and thus cannot be tampered. Unfortunately, as mentioned in Section 4.2.1, 74.3% (168 out of 226) RPs supporting WeChat SSO implement the interaction (Step 6 and 7 in Fig. 7) on their client sides (RP apps), making the exploit feasible again.

Security Impacts and Remedies: The impacts of the vulnerability depend on the property of the injected access token and the authenticator chosen by the RP server.

If the token is issued to the targeted RP, the attacker can log into the RP app as the victim. For example, a network attacker may steal a valid access token. Then, he can invoke the debug API from the IdP to identify which RP the token is issued to and launch the attack on the same one. According to our testing result, 94 (17.1%) tested apps transmit the access token in the plaintext and thus vulnerable.

Table 2: Statistics of the testing results from 550 RP apps

IdPs (# of 3rd-party RP app)	**Augmented Token Replacement**	Profile Vulnerability ‡	Access Token Replacement	Credential Disclosure (Access Token)	Credential Disclosure (Code)	App Secret Disclosure	#of vulnerable RPs †
Sina Weibo (196)	**145**	140	125	48	N/A	88	149 (76%)
WeChat (226)	**119**	98	81	41	23	179	191 (84.5%)
Facebook (128)	**54**	46	24	5	N/A	0	57 (44.5%)
Summary	**318**	284	230	94	23	267	397 (72.2%) *

* 397 out of the 550 RPs (72.2%) are incorrectly implemented.
† One RP app may be susceptible to multiple vulnerabilities, *e.g.*, access token replacement and profile vulnerability, simultaneously.
‡ The scope of the detected Profile Vulnerability is larger than [49], where we also take the user profile from malicious RPs into account.

In the scenario, the RPs that utilize the SSO services from Sina Weibo and WeChat cannot detect the attack so that they must protect the access token well. In contrast, the RPs using the Facebook SSO service can detect the injection by verifying the signature within the id token (Section 2.3), if the signing key is not leaked.

In contrast, if the token is issued to a different RP, *i.e.*, a malicious RP attacker, the impacts rely on the chosen authenticator within the user information (*e.g.*, Step 8 in Fig. 1). If the authenticator is shared among different RPs, *e.g.*, email, the attacker can still steal a benign RP account. Otherwise, the attacker may only forge a malicious RP account with the IdP identities of victims. Using the forged account, the attacker can post malicious content on the RP and hurt the reputation of the victim. In many cases, some information required for registration cannot be directly forged as additional verification is needed, *e.g.*, SMS verification. However, the information returned by IdP is trusted by the RP, thus the attacker can exploit the vulnerability to bypass the verifications and create malicious accounts with valid user profiles (from the IdP).

To mitigate the vulnerability in the situation, the RP server should verify the binding between the received token and itself.

5.3.2 *Discovery of Code Maintenance Failure*

Observations: In addition to the vulnerabilities resulted from the incorrect implementations by RPs, MoSSOT also finds a new one caused by the IdP. According to the protocol specification [21], the code used in the authorization code flow should be short-lived.

However, after analyzing the testing result, we find that an unused code generated more than 100 minutes ago (10 times longer than the claimed value in [44]) will still be accepted by the server. Thus, the IdP server does not maintain authorization code properly.

Security Impacts and Remedies: The vulnerability facilitates an attacker to exploit the stolen code (available through [15, 24, 28]) and log into the RP app as the victim. In the case of Credential Disclosure [47], a network attacker may intercept the code from the RP app to its server (Step 5 in Fig. 7). According to the testing result, around 10% RP apps (23 out of 223 in Table. 2) indeed disclose the code in the plaintext and thus is susceptible to the vulnerability.

It is the duty of IdP to fix the vulnerability, where its server needs to shorten the validity period of codes and reject expired ones.

5.3.3 *Improved Detection of App Secret Disclosure*

Besides the three known vulnerabilities discussed above, MoSSOT also detected App Secret Disclosure [11, 47].

The app secret is the identity proof issued by the IdP to RP and required in the critical requests from the RP server to the IdP server. However, as mentioned in Section 4.2.1, some RPs tend to customize the logic on the client side (RP app). Then, the attacker can steal the secret and use it to impersonate as the benign RP/ IdP for cheating the other party. For example, in the Facebook case, the app secret is used as the signing key of the id token (in Fig. 8). Consequently, the attacker can impersonate as the benign IdP and forge a valid signature to cheat the RP server into the wrong authentication.

Different from [11, 47], we extended MoSSOT and utilized a hybrid method to improve the accuracy of detecting the known vulnerability. Readers may refer to Appendix. C for more details about the extension. Overall, 267 RP apps (48.5%) leak the app secret and none of the Facebook apps is vulnerable, which may attribute to the explicit warning in its documentation.

6 DISCUSSION AND FUTURE WORK

As reflected in Table. 1, MoSSOT encounters several obstacles in the test. In the section, we elaborate the open issues in different testing phases and give some proposed solutions.

6.1 Misclassification in Dataset Screening

Misclassification in the screening (Section 5.1) is unavoidable. We apply conservative strategies to limit the false negative rate, as UI Explorer (Section 4.1.2) can eliminate false positives. We run the Screener against the whole dataset and randomly selected 200 passed apps to evaluate its performance. We manually examined each of them to get the true positives and analyzed the rest for failure investigation. The result is shown in Table 3.

The high false positive rate, 84 out of 200 (42%), is mainly due to the fact that many apps include SSO SDK without using it. However, it is possible to improve the accuracy so that UI Explorer will not waste time on those apps. One potential solution is to extract the call flow graph from the APK and check the usage of SSO SDK.

6.2 UI Automation Failure

The most challenging part in UI automation is the exploration. The step is crucial as it feeds input to Test Engine (Module B in Fig. 3). Based on the ground truth in Table 3, we evaluate and compare the performance of UI Explorer with both Algorithm I and Algorithm II (described in Section 4.1.2). The success rate and average running time are summarized in Table 3. The result indicates the choice between Algorithm I and Algorithm II is actually a tradeoff between false negative rate and efficiency. In our experiments, the runtime overhead of Algorithm II is mainly caused by the penalty of false positives. For example, if there is a button appearing to be a login

Table 3: Performance evaluation dataset: 200 sample apps that passed Screener

Apps that pass manual SSO test * 73/200	Performance of Algorithm I & II		
	Algorithm I (LKS)	Algorithm II (DFS)	Both Algorithms failed
	47/73 (64.4%), $\bar{t}$ = 236s †	57/73 (78.1%), $\bar{t}$ = 376s	14/73 (19.2%)
Apps that cannot pass manual SSO test 127/200	Reasons that manual SSO test failed		
	SSO not integrated	Launch failure ‡	Update required
	84/127 (66.1%)	33/127 (26.0%)	10/127 (7.9%)

* The manual SSO test was done in an emulator. † $\bar{t}$ is the average running time per app. ‡ The failure can be caused by the emulator or the app itself (*e.g.*, unable to connect to its server).

Table 4: Statistics of the failure cases in Test Learner

Failure Reasons	#Cases (percent)	Fixable with manual config.	Fixable with RP's support
App Error	18 (30%)	×	×
RP Account Settings	14 (23.3%)	✓	✓
Failure to Extract User Login Status	27 (45%)	×	✓
Captcha Required	1 (1.7%)	×	✓

button but turns out it is not, then both algorithms will choose it first. After clicking the button, Algorithm I will cut off directly as there is no targeted keyword in the new page. However, the other will try every new button because its core is depth-first search.

The overall accuracy of UI Explorer is reasonable and failure cases (19.2%) are mainly due to the imperfection of Noise Reducer. Our current design cannot handle some corner cases, *e.g.*, UI widgets with no identifiable characteristics. Some of them can be fixed with one-time human assistance. For the purpose, we developed a tool which enables users to navigate the app to the login page and take snapshots of the emulator via web browsers. Then, MoSSOT simply reloads that snapshot to reach the login page during the test. Meanwhile, we plan to apply static analysis on the app (APK) to extract input constraints and solve them to assist the UI exploration.

Besides, many apps (26.0%) in the sample set cannot be launched. For example, their backend servers are no longer maintained, which is unfixable. Some others refuse to run in the emulator and can be tested with proper setup, as MoSSOT can execute on real devices.

6.3 Obstacles in the Learning Phase

According to Table. 1, only 27% of the apps could pass the whole testing phases after the UI exploration, which is mainly caused by the failures in learning app-specific SSO implementations (Section 4.2.2). We manually analyzed 60 failure cases, which can be categorized into four types as shown in Table. 4.

App Error: Although the tool did not tamper any message in the step, two of the apps crashed frequently. In the other cases, the backend RP servers responded with error messages.

RP Account Settings: In the category, the RP accounts required special settings beforehand, *e.g.*, phone number binding, so that the tool could not finish the whole SSO process.

Failure to Extract User Login Status: MoSSOT relies on the RP authentication response to identify the RP login status (Section 4.3.2). However, MoSSOT may fail to learn it as these RPs use customized protocols so that our tester cannot capture the message. At the moment, we do not have a proper solution to the issue as there does not exist a uniform method to parse the TCP messages.

Captcha Required: We encountered one app requiring Captcha verification, which our UI automation module could not handle.

As indicated in Table. 4, RP Account Settings can be fixed with the one-time manual configuration so that the success rate at the stage can increase to around 44.2%. Besides, Failure to Extract User Login Status and Captcha Required may be solved given the support from the RP, where the rate can increase to around 78.1% further.

6.4 Speeding up the Execution of Test Cases

Most apps could complete the test once they passed the previous phases due to the special handling mentioned in Section 4.5. However, our tester still suffers a large time cost, which is mainly caused by network delay and UI navigation (Section 4.1).

In the normal flow, the RP server will not reply the authentication request (*e.g.*, Step 6 in Fig. 1) to the app until the handshake between the servers (*e.g.*, Step 7 and 8 in Fig. 1) is finished. Since one of our targets is to find the vulnerabilities in the blackbox servers, the testing must be online and the network delay cannot be avoided.

Besides, the UI Navigation is also time-consuming because every widget in the UI path needs triggering in each round. Worse still, once noise appears, *e.g.*, advertisement popup, Noise Reducer needs to take more time to recover the app to the normal state.

There are two possible solutions to increase the speed. The first one is to use the snapshot functionality from the Android Emulator [18]. Then, the tool can take the snapshot of the RP login page after the UI exploration and reload it in the test, whose time cost is expected to be lower and not affected by the noise.

The second solution is to skip the interactions between the IdP app and IdP server, *e.g.*, Step 2 to 4 in Fig. 1. From the perspective of the RP developers, they are only interested in assessing their own SSO deployments. Then, we may prepare some valid data, *e.g.*, access token and user profile, and configure the proxy beforehand. Consequently, once the requests from the IdP app is detected, Proxy will impersonate as the real IdP server and respond immediately.

7 RELATED WORK

UI automation for Android app testing. Many projects have been done in recent years on automated Android app testing. Most of them aim at exploring the app with larger coverage. In contrast to traditional random exploration used by Monkey [19], they apply more systematic strategies. GUIRipper [2], SwiftHand [12], PUMA [20], and DroidBot [30] crawl an app and dynamically build a finite state machine to represent the app's UI model. Among them,

Table 5: Comparison with Previous Work on SSO Testing

Selected Work	Level of Automation	Scope of Study	Target
Zhou et al.(SSOScan) [53]	Automatic	1660 websites	4 specific vulnerabilities
Sun et al. [43]	Semi-automatic	96 websites	5 specific vulnerabilities
Shernan et al. [39]	Automatic	10000 websites	Assessing the usage of the *state* variable
Li et al. [29]	Manual	103 websites	3 specific vulnerabilities
Shehab et al. (OAuthManager) [38]	Automatic	430 Android apps	3 specific vulnerabilities *
Wang et al.(AuthDroid) [47]	Semi-automatic	100 Android apps	6 specific vulnerabilities †
Wang et al. [46]	Semi-automatic	79 websites, 85 Android apps & 77 iOS apps	5 specific vulnerabilities ‡
Our work (MoSSOT)	Automatic	550 Android apps	General model-based testing (detecting 4 known and 2 unknown vulnerabilities)

* The vulnerabilities are about the improper usage of Android WebView [17] and is out of the scope of our study.
† [47] also takes the inter-app (*e.g.*, Step 1 & 5 in Fig. 1) and server-to-server (*e.g.*, Step 7 & 8 in Fig. 1) communication into account.
‡ [46] considers the usage of access token (instead of id token) in OIDC (Section 2.3) as a vulnerability, which may not be exploitable.

GUIRipper allows the tester to configure inputs to be used during exploration. SwiftHand uses an exploration strategy that can minimize the app restarts. PUMA provides a framework to combine model-based exploration with random monkey inputs. Instead of dynamically building the model, some other tools, *e.g.*, A^3E [4] and FraudDroid [13] extract activity transition graph beforehand with static code analysis to guide the UI testing. More recently, researchers start to explore advanced strategies like a stochastic model [42] and machine learning [9, 36]. However, all these tools try to construct a map of every activity in an app for exploration while our task is to look for SSO login interfaces. Our algorithms eliminate the overhead of map construction to achieve better efficiency. Whatsmore, all mentioned tools lack the ability to replay their recorded UI path reliably as indicated in [27]. To address the challenge, we packaged three modules, *i.e.*, Explorer, Navigator and Noice Reducer, into our work to make UI path replay possible.

The projects with more relevant goals to ours are Brahmastra [7] and AuthScope [54], both focusing on driving apps to the targeted activity. While [7] looks for activity transition paths with static analysis, [54] implements prioritized DFS for targeted UI exploration. Nevertheless, [54] only works with Facebook SSO login. In contrast, our work is more extensible and is capable of handling multiple IdPs. Although both of our work and [54] utilize DFS for UI exploration, [54] only uses keywords and action bindings as prioritization criteria, while our work calculates a score for each element based on more attributes and a smarter algorithm (Section. 4.1.3) for better accuracy and efficiency. Besides, our work also supports LKS. Since [54] is close-sourced, we did not manage to make a comparison between their DFS and our LKS algorithm. However, according to the experiment result (in Table. 3), LKS is more efficient in finding targeted activity than DFS. Besides, as LKS is orthogonal to DFS, it helps to increase the success rate further.

OAuth security studies from the protocol perspective. RFC specifications [21, 31] discuss the security considerations and threat models for OAuth 2.0. Focusing on the classical web attacks like XSS, CSRF, and the intentional attacks specifically designed for OAuth, these standards hope to exclude these common pitfalls. Hu *et al.* [22] present the App Impersonation attack. Besides, under the assumption that the TLS is utilized properly, the authorization code flow has been proven to be secure cryptographically [10].

On the other hand, the formal method is widely adopted by the previous work to assess OAuth Security. Bansal *et al.* [6] model different configurations of the OAuth protocol and analyze them by ProVerif [8], which leads to the discovery of *Token Redirection Attack* and *Social CSRF Attack*. Similarly, AuthScan [5] performs a whitebox code analysis and a blackbox fuzzing to extract the protocol specifications from real implementations and find 7 security flaws. Following their work, Fett *et al.* [15] use an expressive FKS model to perform an extensive analysis of all four grant flows.

These studies prove/ improve the security for OAuth 2.0 from the viewpoint of protocol design. However, since the OAuth protocol was initially designed to serve the authorization need for websites, the focus of the paper, namely the authentication services on mobile platforms, is thus not considered by the studies.

Analyses of mobile OAuth-based SSO systems. In contrast to the wide deployment, there are few security analyses on the mobile OAuth-based SSO systems. Chen *et al.* [11] show how practical OAuth system may fall into the common pitfalls when utilizing the OS-provided components, *e.g.*, Intent, improperly. Shehab *et al.* [38] reveal 3 vulnerabilities in WebView, which affect OAuth security. Ye *et al.* [51] utilize the model checking method to analyze the OIDC-like protocol implemented by Facebook on Android platform and discover a problem on unauthorized storage access. Wang *et al.* [46, 47] perform static code analysis and dynamic analysis on the real-time network messages, leading to the detection of several vulnerabilities across both Android and iOS platforms. Using similar approaches, Yang *et al.* [49] find Profile Vulnerability.

Previous work relies on the manual discovery of vulnerabilities, which is not scalable. Compared to the state-of-the-art, MoSSOT can discover vulnerabilities automatically.

SSO security testing tool. Motivated by the prevalence of vulnerabilities in real-world SSO systems, large-scale security testing has received increasing attention. Sun *et al.* [43] build a semi-automatic tool to test specific vulnerabilities for 96 applications. SSOScan [53] investigates five specific attacks on Top 1600 Facebook websites. Shernan *et al.* [39] analyze the known CSRF attack on 10,000 websites by checking the existence of *state*. Li *et al.* [29] report the security quality of 103 Google-enabled RP websites.

Nevertheless, all the work mentioned so far only studies the specifications/ implementations of SSO in the web applications, where

the interactions for secure authentication are well specified. In contrast, we focus on the mobile platform, where such interactions of our interest are error-prone and overlooked.

The projects most relevant to ours are [50] and [54]. [50] also utilizes the model-based testing to assess real-world SSO deployments, but targets web applications instead of mobile apps, where the situation is not so complicated. For example, [50] may enter any state in their model by constructing a proper URL request. In contrast, we are incapable of maintaining the app state directly and have to rely on UI to trigger SSO-related network messages. On the other hand, although our method is similar to [54], our target is the vulnerabilities within the SSO (authentication), while [54] focuses on the authorization issues after the authentication process.

8 CONCLUSION

In this paper, we present an automated blackbox security testing tool, MoSSOT, to systematically test the implementations of SSO by the RPs/ IdPs as well as their backend servers. We implement the tool and perform the test on 550 RP apps. The tester identified one previously-unknown vulnerability and a new variant, in addition to four known ones. All of them can break the authentication of the RP apps and lead to privacy leakage of the victims.

We have open sourced MoSSOT at [34] and plan to extend it for other protocols, *e.g.*, mobile payment protocols, in the long run.

ACKNOWLEDGEMENT

We thank our shepherd Dr. Guangdong Bai and the anonymous reviewers for their valuable comments and suggestions. We also thank Yihui Zeng, Ronghai Yang, Zhuowei Zhong, Guanchen Li, and Chakman Li for their contributions in the development of MoSSOT. The work is supported in part by the ITF of HK (project#ITS/216/15), the CUHK TBF (project#TBF18ENG001), the CUHK PIEF (project#3133043), and the 2018 Facebook/USENIX Internet Defense Prize.

REFERENCES

[1] 2017. Culebra. https://github.com/dtmilano/AndroidViewClient/wiki/culebra
[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *ASE12*. ACM.
[3] Apkpure. 2017. Apkpure. https://apkpure.com/.
[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *ACM Sigplan Notices*, Vol. 48. ACM.
[5] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. 2013. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS13*.
[6] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. 2012. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *CSF12*.
[7] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components.. In *USENIX14*.
[8] Bruno Blanchet. 2014. The ProVerif homepage. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/
[9] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *MOBILESoft18*. ACM.
[10] Suresh Chari, Charanjit S. Jutla, and Arnab Roy. 2011. Universally Composable Security Analysis of OAuth v2.0. Cryptology ePrint Archive, Report 2011/526.
[11] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. OAuth demystified for mobile application developers. In *CCS14*.
[12] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM Sigplan Notices*, Vol. 48. ACM.
[13] Feng Dong, Haoyu Wang, Yuanchun Li, Yao Guo, Li Li, Shaodong Zhang, and Guoai Xu. 2017. FrauDroid: An Accurate and Scalable Approach to Automated Mobile Ad Fraud Detection. *arXiv preprint arXiv:1709.01213* (2017).
[14] Facebook. 2017. Facebook SSO developer document. https://developers.facebook.com/docs/facebook-login/.
[15] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. *CCS16* (2016).
[16] Genymotion. 2017. Genymotion. https://www.genymotion.com/
[17] Google. 2017. Android webview. http://developer.android.com/reference/android/webkit/WebView.html
[18] Google. 2017. AVD. https://developer.android.com/studio/run/emulator.
[19] Google. 2017. Monkey. http://developer.android.com/tools/help/monkey
[20] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys14*. ACM.
[21] Dick Hardt. 2012. The OAuth 2.0 authorization framework.
[22] Pili Hu, Ronghai Yang, Yue Li, and Wing Cheong Lau. 2014. Application impersonation: problems of OAuth and API design in online social networks. In *COSN14*.
[23] Jonathan Jacky. 2011. PyModel: Model-based testing in Python. In *SciPy11*.
[24] Wang Jing. 2017. *Covert Redirect Vulnerability*.
[25] M Jones and Dick Hardt. 2012. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Technical Report. RFC 6750, October.
[26] Michael Jones, Paul Tarjan, Yaron Goland, Nat Sakimura, John Bradley, John Panzer, and Dirk Balfanz. 2012. JSON Web Token (JWT). (2012).
[27] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for Android: are we there yet in industrial cases?. In *ESEC/FSE17*. ACM.
[28] Wanpeng Li, Chris J Mitchell, and Tom Chen. 2018. *Your code is my code: Exploiting a common weakness in OAuth 2.0 implementations*.
[29] Wanpeng Li and Chris J. Mitchell. 2016. Analysing the Security of Google's implementation of OpenID Connect. In *Proceeedings of DIMVA16*.
[30] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *ICSE17*. IEEE.
[31] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. 2013. OAuth 2.0 threat model and security considerations.
[32] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *ESEC/FSE13*. ACM.
[33] mitmproxy. 2017. Man in the Middle Proxy. https://mitmproxy.org/
[34] MoSSOT. 2019. MoSSOT. https://github.com/MoSSOT/MoSSOT.
[35] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppsPlayground: automatic security analysis of smartphone applications. In *ACM CODASPY13*.
[36] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2018. Automation of Android Applications Functional Testing Using Machine Learning Activities Classification. In *MOBILESoft18*. ACM.
[37] Natsuhiko Sakimura, J Bradley, M Jones, B de Medeiros, and C Mortimore. 2014. OpenID Connect core 1.0. (2014).
[38] Mohammed Shehab and Fadi Mohsen. 2014. Towards Enhancing the Security of OAuth Implementations In Smart Phones. In *IEEE MS14*.
[39] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. 2015. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *DIMVA15*. Springer.
[40] Sina. 2017. Sina Developer Documentation. http://open.weibo.com/wiki/
[41] Softonic. 2017. Softonic. https://bit.ly/2DUAjhp.
[42] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based gui testing of android apps. In *ESEC/FSE17*. ACM.
[43] San-Tsai Sun and Konstantin Beznosov. 2012. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *CCS'12*.
[44] Tencent. 2017. WeChat SSO developer document. https://open.weixin.qq.com.
[45] Wandoujia. 2017. Wandoujia App Market. https://www.wandoujia.com/.
[46] Hui Wang, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. The Achilles Heel of OAuth: A Multi-platform Study of OAuth-based Authentication *(ACSAC '16)*.
[47] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. 2015. Vulnerability Assessment of OAuth Implementations in Android Applications. In *ACSAC15*. ACM.
[48] Xposed. 2017. Xposed Module Repository. https://repo.xposed.info
[49] Ronghai Yang, Wing Cheong Lau, and Shangcheng Shi. 2017. Breaking and Fixing Mobile App Authentication with OAuth2.0-based Protocols. In *ACNS17*.
[50] Ronghai Yang, Guancheng Lee, Wing Cheong Lau, and Kehuan Zhang. 2016. Model-based Security Testing: an Empirical Study on OAuth 2.0 Implementations. In *ASIACCS 2016*.
[51] Quanqi Ye, Guangdong Bai, Kailong Wang, and Jin Song Dong. 2015. Formal Analysis of a Single Sign-On Protocol Implementation for Android. In *ICECCS15*.
[52] E. YOO. 2017. Technode. https://bit.ly/2Zi1JVn.
[53] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *USENIX14*.
[54] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *CCS17*.

A MORE ON MOBILE SSO PROTOCOLS

A.1 Protocol Flow of Authorization Code Flow

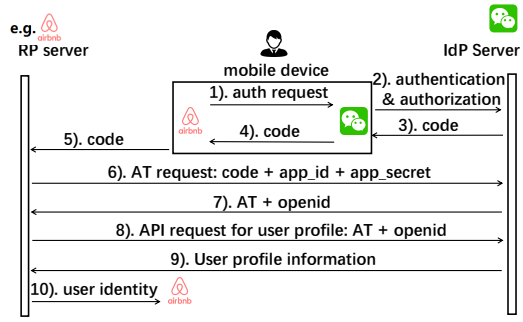

Figure 7: Authorization code flow of OAuth 2.0 for mobile platforms

A.2 Protocol Flow of OpenID Connect

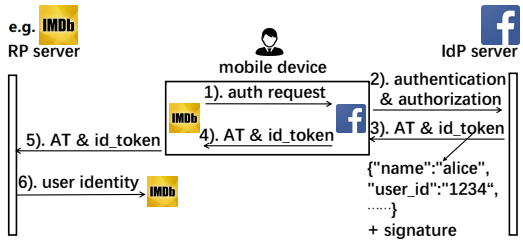

Figure 8: Implicit flow of OpenID Connect for mobile platforms

B MORE ON UI EXPLORATION

B.1 Details of Algorithm I

Algorithm 1: Pseudocode for Algorithm I (LKS) of Explorer

```
initialize ladder[n] to be the predefined list of keywords;
initialize UI path as an empty list;
Function LKS(ladder)
    i ← 0 ;
    page ← source code of current activity;
    if page is the desired destination then
        // end of recursive call
        return recorded UI path;
    while i < len(ladder) do
        foreach keyword in ladder[i] do
            if an element in page matches keyword then
                click on the element;
                append element to UI path;
                ladder ← ladder [0..i] ;
                // recursive call
                return LKS(ladder);
        i ← i + 1;
    return cannot find path to the destination;
```

To generate the list of keywords in Algorithm I (LKS), we manually logged into 100 Android apps via SSO for each studied IdP. At the same time, a self-developed XPosed [48] module records the information of the triggered widgets and outputs related keywords. Then, we prioritized the keywords into different levels according to their occurrence frequency. On average, there are around 25 keywords for each IdP and 5 keywords for each level. One example of the prioritized keyword list for Facebook is listed as follows, where widgets with the keywords on the upper level have a higher probability to link to the SSO login page.

Level 5 "*facebook*", "*fb*"
Level 4 "*third-party login*", "*login*", "*log in*", "*signin*", "*sign in*", "*register*"
Level 3 "*username*", "*avatar*", "*nick*", "*profile*", "*account*", "*personal info*", "*user*", "*head*", "*edit*"
Level 2 "*setting*", "*option*"
Level 1 "*logged out*", "*personal*", "*mine*", "*drawer*", "*menu*", "*home*"

Based on the list, Algorithm I will search for the keyword from top to down and trigger the associated widget during UI exploration.

B.2 Details of Algorithm II

To utilize the DFS algorithm, we model the Android UI as a weighted directed graph $G = (V, E)$. There are two types of vertices: $V_e \subset V$ is the set of element vertices, and $V_p \in V$ is the set of page vertices. Similarly, we define two subsets representing two types of edges. Particularly, $E_{pe} \in E$ is the set of page-to-element edge, modeling the action when the user interacts with an element in a page. $E_{ep} \in E$ is element-to-page edges, modeling the behavior of jumping to a new page after an element is tapped. A simplified graph representation of a toy app with four pages is shown in Figure 9 to give readers a concrete idea. In the figure, larger circles with labels are page vertices, and those smaller circles are element vertices. We are only interested in the elements that lead to different pages and ignore those edges representing intra-page jump. Therefore, in our graph model, vertices $v \in V_e$ are all disconnected, same for $v \in v_p$. Also, $\forall v \in V_e$, the outdegree $deg^+(v) \in \{0, 1\}$, meaning that an element can either lead to a particular page or nowhere. The indegree $deg^-(v) = 1$, showing that an element can only appear on one page. For a page vertex $v \in V_p$, we have $deg^-(v) \in \mathbb{N}^+$, since different elements can point to the same page. Vertex p_0 and p_1 in Figure 9 are examples of this case. The graph can be cyclic, the dashed line in the figure shows an example when the back edge leads to a cycle. Weights are assigned to page-to-element edges to represent the preference among elements in the same page. Under this graph model, two major challenges for the DFS algorithm are loop cutting and prioritizing.

Loop Cutting. Since the graph can be cyclic, we need to detect and cut the loop when running the DFS. We use the standard cycle detection method, namely keeping track of visited vertex to detect the back edge. The actual challenge for this is the ability to detect identical elements in different contexts, as no single attribute is unique for UI elements. Our solution is combining several attributes to determine if two elements are identical. We begin by checking whether all of the string type attributes (*id*, *text*, and *desc*) are equal. If so, we further calculate the perceptual hash to quantify the similarity of their screenshot image. If the similarity ratio is above a certain threshold, we consider these two elements to be identical.

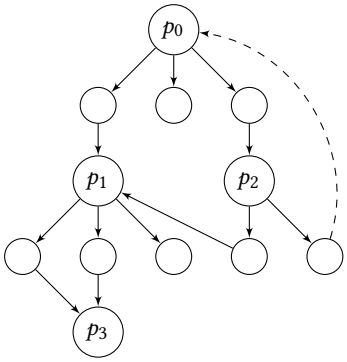

Figure 9: Graph representation of Android UI

- Vertices with labels are V_p, while the others are V_e.

Prioritizing. Each page vertex has outgoing edges linking to element vertices, displaying the relation that a group of elements contained in the same page. When DFS arrives at a page, it is more efficient if some heuristics can be applied to first try those more promising elements rather than choosing randomly. To achieve that, we define a scoring function $F : E_{pe} \mapsto [0, 1]$ to calculate a weight for each of page-to-element edge. More specifically, F is a weighted sum of individual scoring functions for each property of the element the edge points to, namely $F(e) = \sum w_i f_i(e)$. Properties we select as scoring components are *id*, *desc*, *text*, *clickable*, *type*. Weights for each property is set empirically. For instance, *clickable* is a strong indicator comparing to *type*, so we put more weights on it. For individual scoring function, we would like to mention two of our major innovations here.

- *String score function.* We introduce a unified string score function for string type properties (*id*, *desc*, *text*). First, we build a weighted keyword database based on the same keyword statics we used in Algorithm I. Weights are assigned based on the frequency of keywords. Given a string-type property of an element, we search for all substring matches in the keyword database, for each of them we get a raw score s, which is equal to the keyword weight. Instead of directly using this raw score, we calculate an adjusted score $s' = cs$ based on the length of the property string and the matched substring. Finally, the maximum adjusted score among all matches is returned. The adjustment coefficient c is defined by

$$c = \frac{L - x}{Lk^x}, \text{ where } x = \min\{L, l_s - l_m\}$$

In this equation, l_s, l_m represents the length of the property string and matched substring respectively, L is maximum acceptable string length, and k is a parameter controlling the steepness of decrease for the function. We fixed its value to be 1.1. The intuition behind the adjust function is that a string with longer unmatched part usually has less semantic similarity to the matched keyword. For example, a UI widget with text *"login now"* looks more promising as a login button compared to a UI widget with the text *"you can login with email or mobile number"*, even though both of them match the keyword *login*. Here, we set L as an upper bound because widgets with very long text are mostly descriptive, *e.g.*, articles, which is not interactive. Lastly, we did not use linear function (*i.e.*, setting $k = 1$) since the semantic vanishes sharply when the unmatched length increases. Figure 10 shows the curves of the adjust function with different k.

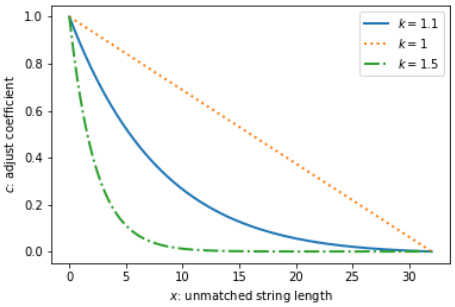

Figure 10: String score adjust function, when $L = 32$

- *Identify clickable elements.* Clickable elements often lead to new pages. Android UI element natively has an attribute called *clickable* which defines whether it reacts to click events. However, elements can overlay on each other and present to users on the same surface. From the user's point of view, an element is clickable if some events can be triggered when tapping on its area, regardless of its *clickable* property. Therefore, we call an element with *clickable* property to be *explicitly clickable*, while element not declared *clickable* but appears to be clickable to users as implicitly clickable. To identify all *implicitly clickable* elements, we calculate the union of areas covered by *explicitly clickable* elements and then mark other elements intersecting with the area to be *implicitly clickable*.

C EXTENSION FOR DETECTING APP SECRET DISCLOSURE

Originally, MoSSOT detected the vulnerability indirectly by monitoring the network traffic through Proxy, where the appearance of critical requests (involving app secret) indicates the vulnerability (Section. 4.3.2). However, the method suffers false negatives as RP developers may hardcode the secret in their APKs without using it. Thus, we extended MoSSOT for better detection accuracy.

Similar to the method in [47], the extended tool forges the critical requests (appended with the app secret) to the IdP server and confirms the leaked secret based on the response content, where the potential app secret is available from the decompilation result of the APK, *i.e.*, Step 1(b) in Fig. 3. Typically, the kind of requests involves three parameters, namely app id, app secret, and a credential (*e.g.*, access token), and the IdP server verifies them in the same order. Thus, once the app id is correct, we can get the judgment of the chosen app secret from the IdP server. Previous work [47] attempted to obtain the app id via reverse engineering and might fail due to the protection on the APK. In contrast, our tool extracts the app id from the reference network traffic (Step.4 in Fig. 3), which always appears in the interactions between the IdP app and its server (*e.g.*, Step 3 in Fig. 1). Therefore, our tool is expected to have higher accuracy in detecting the vulnerability.

With the extension, MoSSOT detected 187 vulnerable apps and 99 of them were not captured by the original method.