An Empirical Study of SMS One-Time Password Authentication in Android Apps

Siqi Ma
CSIRO
siqi.ma@csiro.au

Runhan Feng, Juanru Li
Shanghai Jiao Tong University
{fengrunhan,jarod}@sjtu.edu.cn

Yang Liu
Xidian University
xdly9491@gmail.com

Surya Nepal, Diethelm Ostry
CSIRO
{surya.nepal,diet.ostry}@csiro.au

Elisa Bertino
Purdue University
bertino@purdue.edu

Robert H. Deng
Singapore Management University
robertdeng@smu.edu.sg

Zhuo Ma
Xidian University
mazhuo@mail.xidian.edu.cn

Sanjay Jha
University of New South Wales
sanjay.jha@unsw.edu.au

ABSTRACT

A great quantity of user passwords nowadays has been leaked through security breaches of user accounts. To enhance the security of the Password Authentication Protocol (PAP) in such circumstance, Android app developers often implement a complementary One-Time Password (OTP) authentication by utilizing the short message service (SMS). Unfortunately, SMS is not specially designed as a secure service and thus an SMS One-Time Password is vulnerable to many attacks. To check whether a wide variety of currently used SMS OTP authentication protocols in Android apps are properly implemented, this paper presents an empirical study against them. We first derive a set of rules from RFC documents as the guide to implement secure SMS OTP authentication protocol. Then we implement an automated analysis system, AUTH-EYE, to check whether a real-world OTP authentication scheme violates any of these rules. Without accessing server source code, AUTH-EYE executes Android apps to trigger the OTP-relevant functionalities and then analyzes the OTP implementations including those proprietary ones. By only analyzing SMS responses, AUTH-EYE is able to assess the conformance of those implementations to our recommended rules and identify the potentially insecure apps. In our empirical study, AUTH-EYE analyzed 3,303 popular Android apps and found that 544 of them adopt SMS OTP authentication. The further analysis of AUTH-EYE demonstrated a far-from-optimistic status: the implementations of 536 (98.5%) out of the 544 apps violate at least one of our defined rules. The results indicate that Android app developers should seriously consider our discussed security rules and violations so as to implement SMS OTP properly.

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359828

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Multi-factor authentication*; Software reverse engineering;

KEYWORDS

Authentication Protocol; Mobile Application Security; One-Time Password Authentication; Vulnerability Detection

ACM Reference Format:
Siqi Ma, Runhan Feng, Juanru Li, Yang Liu, Surya Nepal, Diethelm Ostry, Elisa Bertino, Robert H. Deng, Zhuo Ma, and Sanjay Jha. 2019. An Empirical Study of SMS One-Time Password Authentication in Android Apps. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3359789.3359828

1 INTRODUCTION

Many Android apps commonly use password authentication protocols to verify user identity (i.e., authenticating the user with the combination of the username and a static password [29]). However, in recent years, many security breaches

led to large-scale password leakage[1]. Moreover, users usually choose weak passwords (e.g., the infamous "123456") because a secure password is often too complex to remember. Because of such issues, a single password authentication becomes vulnerable to attacks such as brute force or dictionary based search, and thus is not suitable for user login protection.

To address the above security issues, one-time password (OTP, also known as dynamic password) authentication is used. Referred to as two-factor authentication, the enhanced validation requires the user to provide both a static password and a dynamic OTP. The OTP here is used to protect validation systems against typical attacks because attackers must determine and consume each OTP before the legitimate user can do so for each authentication session. Typically, an OTP is generated through a security token or retrieved from the server via a secure channel. For example, Google and Apple use both static passwords set by users and dynamic passwords generated by mobile authenticators (e.g., the Google Authenticator [20]). Another case is the *CITI Bank* app, which requires both the static password and an additional dynamic password generated by a portable security token [5, 21].

Designing a secure OTP authentication protocol is, however, challenging and error-prone [22]. The security of OTP authentication is regulated by a number of RFC documents and a secure OTP authentication must satisfy several security requirements. A crucial requirement is how to generate and deliver the OTP securely. For an Android app, it is often too heavyweight and inconvenient to deploy a portable security token or a mobile authenticator. Therefore, generating and transferring OTP through the short message service (SMS) becomes the most prevalent implementation[2].

We observed that although several techniques have been proposed to analyze designs and implementations of traditional password authentication in Android apps [6, 30, 48], seldom studies considered SMS OTP authentications and barely analyzed the implementation security of them. In this paper, we focus on security requirements of the SMS OTP authentication protocol (hereinafter, we refer it as **OTP authentication protocol**) and conduct an empirical evaluation of the security of user validation systems which implement OTP authentication protocols. The purpose of our study is two-fold: (1) investigating the OTP authentication protocol in the Android ecosystem, and (2) identifying apps which implement vulnerable OTP authentication. We focus on Android apps because Android is the most widely used mobile operating system [8] and a large number of Android apps adopt OTP authentications.

[1]Massive breach leaks 773 million email addresses and 21 million passwords https://www.cnet.com/news/massive-breach-leaks-773-million-emails-21-million-passwords/

[2]https://www.quora.com/Why-do-companies-use-SMS-and-OTP-to-verify-the-mobile-number-doesnt-one-suffice

Our goal is to check how well an OTP implementation (including the server side and the app side) complies with a set of OTP authentication protocol rules (hereinafter, it is referred to as **OTP rules**), which we derive from the best practices outlined in a number of RFCs [3]. Since these rules are recommended as the best practices for implement secure OTP protocols, we propose a novel analysis system, AUTH-EYE, which determines whether OTP implementations violate our defined OTP rules. AUTH-EYE locates and executes login via an app to trigger the relevant functionalities in server's validation system and then examines server behaviours (i.e., server requests and responses). Unlike previous code-based approaches using semantic executions and code dependencies [49, 52], AUTH-EYE only relies on limited app code information (i.e., class names and function names). Moreover, since source code of remote servers is often not available, AUTH-EYE treats each remote server as a black box and thus only checks its authentication system by analyzing the server responses.

We applied AUTH-EYE to assess 3,303 popular Android apps, collected from the top 2 Android app markets: Google Play and Tencent marketplaces. We observed that there are three types of login schemes, only password authentication, only OTP authentication, and two-factor authentication with both password authentication and OTP authentication. Since we only focus on OTP authentication in this paper, AUTH-EYE found 544 apps implemented OTP authentication protocols in total. For the apps with two-factor authentication, we manually registered an account and filled in the corresponding username and password in each app. Surprisingly, AUTH-EYE detected that OTP authentication protocols in 536 (98.6%) out of the 544 apps violate at least one of our defined OTP rules, and only eight (1.4%) apps implement their OTP authentication protocols with all OTP rules satisfied. This indicates that developers may not be aware of the OTP security recommendations outlined in the RFCs, and AUTH-EYE can be used to help them implement more secure OTP authentication protocols.

Contributions: Our contributions are:

- We derived a set of OTP rules that developers should follow to implement secure OTP authentication protocol. Those rules are summarized from RFC documents and then used to check implementations of OTP authentication protocols in remote servers.
- We proposed a novel analysis system, AUTH-EYE, to automatically evaluate protocol implementations. In particular, AUTH-EYE conducts an efficient code analysis to locate login *Activities* in apps, requiring only

[3]Protocols defined in RFC: RFC 4226 [35], RFC 2289 [22], RFC 6238 [36], RFC 1750 [14] and RFC 4086 [15]

limited semantic information. It also examines the remote validation system by only analyzing the network (and SMS) requests and responses rather than source code on server sides.
- We tested 3,303 real-world Android apps with AUTH-EYE and found 544 apps adopt OTP authentication, and AUTH-EYE reported that a large notion of those apps violates at least one of our defined OTP rules.

2 DEFINITIONS AND OTP RULES

In this section, we introduce the OTP authentication protocols, and explain OTP rules for designing and implementing OTP authentication protocols properly.

2.1 One-Time Password Authentication Protocol

When a user account is created in OTP authentication protocols, the account is bound to the user by the possession of some information specific to the user, such as a mobile phone number or an email address. At login, an OTP is created for the user who must correctly return it. Android apps often use SMS OTP authentication, where the server generates a pseudo-random value as an OTP and sends it via SMS to the mobile phone number in the user's profile. Such a pseudo-random value is shared only between the server and the user owning the mobile phone. The user then submits the received value to the server for authentication. The unpredictable and unique nature of the pseudo-random value prevents password replay attacks. Two algorithms (namely HMAC-based OTP and time-based OTP) are widely used to generate the one-time password.

2.1.1 HMAC-based One-Time Password (HOTP). The algorithm of HMAC-based one-time password (HOTP) combines an incrementing counter value (C) and a secret key (K) to generate the one-time password. The OTP value generated by the HOTP algorithm is defined as [35]:

$$value = HOTP(K, C),$$

where $HOTP$ is the function:

$$HOTP(K, C) = Truncate(HMAC_H(K, C)).$$

where H is a cryptographic hash function, and the output of the hash function $HMAC_H$ is truncated to a user-friendly size.

An HOTP value with a short length is convenient, but vulnerable to brute-force attacks. To address this problem, RFC 4426 recommends two steps: 1) the maximum number of possible attempts per login session should be set beforehand, and 2) each failed attempt should introduce an additional delay before a retry is permitted. RFC 4426 also suggests that with these protective steps the length of an HOTP value should be at least six digits.

2.1.2 Time-based One-Time Password (TOTP). The time-based one-time password (TOTP) algorithm [36] is an extension of the HOTP algorithm, using elapsed time increments instead of an event counter. Because of human and network latency, the one-time password for each login session must remain valid over a time interval (defined by the time step parameter). Based on RFC 6238, the OTP value generated by the TOTP algorithm is defined as:

$$value = HOTP(K, C_T)$$

where K is a secret key, and C_T is an integer counting the number of completed time steps between the initial counter time T_0 and the current Unix time. Given a time step T_x in seconds, C_T is calculated as:

$$C_T = \frac{(current\ unix\ time - T_0)}{T_x}$$

Due to the network latency, the number of time steps (C_T) calculated by clients and servers may differ and so resulting in different TOTP values. This problem can be addressed by setting the OTP time step T_x to an acceptable size. The OTPs generated anytime within a time step will be the same and will allow the user to login successfully. However, depending on when a login request is made, a server might reasonably accept OTPs from earlier or later time steps. For example, if an OTP is generated near the end of a time step, the user may compute a counter based on the succeeding time step due to latency. To take this into account, the server may accept OTPs computed from time steps +/-1 from its current time step. A larger time step makes the OTP authentication protocol with a TOTP value become more vulnerable because it offers an attacker more time to guess the TOTP value and consume the TOTP value before the valid user does. To balance the security and the usability of this authentication scheme, RFC 6238 recommends setting the size of the time step to 30 seconds.

Furthermore, the server must ensure that sufficient time has elapsed between generating successive TOTP values so that the number of time steps (C_T) has changed.

2.2 Best Practices and Threats for OTP

In this subsection, we first summarize six OTP rules (i.e., rules for secure OTP implementation) according to RFC documents, and then discuss threats against OTP authentications if one or more rules are violated.

2.2.1 OTP Rules. Several RFC documents such as RFC 4226 [35], RFC 2289 [22], RFC 6238 [36], RFC 1750 [14], and RFC 4086 [15] regulate how to securely implement an OTP authentication protocol. We conclude them as six OTP rules

that developers are recommended to follow for a secure OTP implementation.

R1: OTP Randomness–Use a random value as an OTP for authentication.

The server needs a cryptographically strong pseudo-random number generator to generate the OTP value for each login session, as an attacker can exploit any detectable non-randomness in the successive OTPs. Some poor pseudo-random number generators can be identified from the series [14], or values in the sequence may appear periodically. In the worst situation, an implementation of OTP authentication may keep using a static value as the OTP for all authentication sessions.

R2: OTP Length–Generate an OTP value with at least six digits.

The official document RFC 4226 [35] points out that "*the value displayed on the token **MUST** be easily read and entered by the user.*". It requires that the OTP value should also be of reasonable length. Particularly, RFC 4226 indicates that an OTP value of at least six digits could adequately reduce the probability of a successful brute-force attack. In view of both usability and security considerations, OTP values with a length from six to eight digits achieve the required overall performance.

R3: Retry Attempts–Set a limit on the number of validation attempts allowed per login.

RFC 4226 [35] recommend a maximum number of possible attempts for OTP validation. In particular, when the maximum number of attempts is reached, the server must lock out the user's account to defend against a brute-force attack.

R4: OTP Consumption–Only allow each OTP value to be consumed once.

According to the definition of the OTP authentication protocol, each OTP should only be valid for one authentication session.

R5: OTP Expiration–Reject expired OTP values generated by the TOTP algorithm.

Referring to RFC 6238 [36], the OTP value generated in the next time step **MUST** be different. It represents that the OTP value generated by the TOTP algorithm should only be valid for a limited time.

R6: OTP Renewal Interval–OTP values generated by the TOTP algorithm should be valid for at most 30 seconds.

Due to the network latency issue, RFC 6238 [36] recommends that "*A validation system **SHOULD** typically set a policy for an acceptance OTP transmission delay window for validation.*"; thus, a renewal interval is allowed. For the renewal interval, the login validation system achieves a higher usability by allowing for a longer latency, potentially caused by human and network operations. To balance the demands of usability and security, RFC 6238 suggests that the OTP should be renewed every 30 seconds.

2.2.2 Threats against OTP Authentication. We determine whether an implementation of the OTP authentication protocol is secure, in that it should at least meet the following two requirements:

(1) The authentication protocol should not be vulnerable to brute force attacks.
(2) The authentication protocol should be secure against replay attacks.

We observe that if one or more defined OTP rules are violated, the above two requirements may not be satisfied. In the following, we discuss how the violation of our defined OTP rules threats the security of OTP authentication.

- The violation of **R1** indicates that an OTP becomes predictable, and thus the validation systems are vulnerable to replay attacks, allowing attackers to impersonate legitimate users.
- The violation of **R2** indicates that an OTP is of limited length, which is vulnerable to brute-force attacks and may be cracked within a few minutes [11].
- The violation of **R3** indicates that an attacker could easily guess the OTP value through a brute-force attack if unlimited attempts are allowed.
- The violation of **R4** indicates that a validation system allows an OTP to be used multiple times and thus is vulnerable to replay attacks.
- The violation of **R5** indicates that a validation system accepts an expired OTP value and thus allows an unlimited time for an attacker to discover the OTP and consume it before the legitimate user does.
- The violation of **R6** indicates that a validation system provides a long time window for an attacker to crack the OTP.

Moreover, we observe that even though the violation of a single rule might not cause severe security issues, powerful attacks could be launched if multiple rules are violated simultaneously:

(1) Violation of **R1** and any of the other rules. A static OTP value is always vulnerable to replay attacks.
(2) Violation of **R2** and **R3**. For an OTP value with length less than six, an attacker can easily crack the OTP value if the number of validation attempts is unlimited (i.e., vulnerable to brute force attacks).
(3) Violation of **R2** and **R4**. The violation of **R4** allows an attack to reuse current OTP values to launch replay attacks. At the same time, if **R2** is also violated, attackers can easily extract OTP values for further attacks.

(4) Violation of **R4** and **R5**. The OTP validation system is vulnerable to replay attacks by allowing an OTP to be used multiple times. In addition, if the validation system does not set OTP expiration, an attacker is able to launch replay attacks by providing the same OTP value. Even though an OTP is encrypted, the attacker can request for verification by submitting the encrypted format directly.
(5) Violation of **R2**, **R3**, and **R6**. Similar to situation (2), an OTP validation with a larger renewal interval provides an attack enough time to crack an OTP with less than six digits. It is vulnerable to brute force attacks.

3 OTP AUTHENTICATION IN ANDROID

Analyzing OTP authentication protocols implemented in real-world servers require addressing several challenges. Below, we describe the challenges, each followed by our approach to address it.

Challenge 1: How to identify OTP authentication implementations that violation the aforementioned OTP rules without access to the source code? Static code analysis is the most popular technique to locate implementation flaws. However, this technique does not work because server source code is not publicly available. The other technique is to identify functionalities supported by servers and interact with servers. Many attack-based techniques have been proposed based on this approach [25] [27]. However, we cannot apply this approach in our case for ethical reasons.

Our approach 1: Interacting with servers. We search for implementation violations via legitimate interactions with servers. Executing an app triggers the validation functionalities of OTP authentication implemented in the app server. Referring to the OTP rules defined in Section 2, we design experiments to test the functionalities and determine whether the implemented OTP authentication protocol obeys all the OTP rules.

Challenge 2: How to locate the app code that triggers the validation functionalities of OTP authentication? We perform login by executing Android apps to explore the functionalities of OTP authentication at remote servers. Hence, we need to locate OTP login Activities in apps. To find such Activities, we decompile each app and search for functions declaring login Activities. However, developers use a variety of names for these functions, which makes identification more complicated. While we can recognize the login functions by matching data- and control-dependencies with execution patterns, such a strategy does not always succeed because the code decompilation of an app may be incomplete. Thus, we need a broader approach to identify login functions that requires less code information.

Our approach 2: Recognizing OTP login functions semantically. Login Activity declarations, including class information (i.e., class name and name of the extended class) and function information (i.e., function name and argument name) are more complete than other relevant information, such as control-dependencies and API names. Moreover, login Activity declarations reflect which functionalities are included in the class. Although name recognition is challenging, we observe that developers prefer to use similar words to name similar functionalities. This provides a way to identify login-related functions by applying syntax and lexical analysis.

Challenge 3: How to perform login Activities to interact with each server? Once the login Activity declarations are located, we send login requests via the login Activity in each app, triggering the OTP validation system at the remote server. We can manually send login requests, but it does not scale to a large number of apps. To automate the process, we need to design a system that follows the login process precisely. Otherwise, irrelevant services might be triggered, which cause unexpected errors. For example, a button requesting a password reset might be clicked accidentally, possibly switching from the current login page to a password reset page.

Our approach 3: Extracting the position of each widget. After app decompilation, there is a file describing all the involved widgets, including their names, layouts, types, positions, etc. The widget type information helps to identify the widgets used for editing texts and clicking buttons. The widget name implies its purpose, and its layout gives its position. Through this information we can locate the required widgets precisely and execute further operations.

Challenge 4: How to parse the received server messages? Responses from a server are texts containing both useful information (e.g., valid OTP, its expiration, etc.) and irrelevant material. Furthermore, OTP authentication protocols implemented in different servers have different functionalities, which indicates that their responses differ. To deal with this issue, we need a mechanism to systematically process these server messages and recognize the fields containing the useful information.

Our approach 4: Examining altered fields in each message. We identify fields that are altered by comparing multiple responses. Although OTP authentication protocols with different functionalities may give different responses, some essential fields in these responses still follow particular formats. For example, the description of OTP expiration is usually in the format of a decimal followed by a string as “second(s)”, “minute(s)” or “hour(s)”. With some prior information like this, we discover the used formats and furthermore

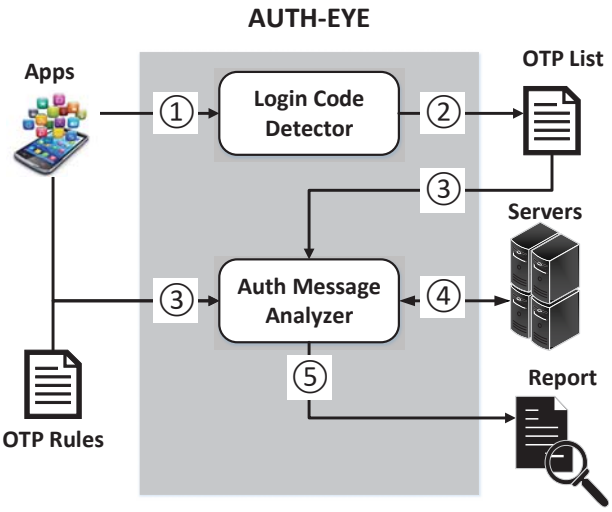

Figure 1: The workflow of AUTH-EYE

find what functionality is implemented by sending multiple requests to the same server.

4 SYSTEM DESIGN: AUTH-EYE

This section describes our automated system, AUTH-EYE, which analyzes Android apps and detects violations of the OTP rules defined in Section 2. AUTH-EYE is built following the approaches outlined in Section 3 to address the critical challenges while examining OTP protocol implementations. AUTH-EYE comprises two components: a **Login Code Detector** and an **Auth Message Analyzer**, and Figure 1 illustrates the system workflow. We next describe the system in detail.

4.1 Login Code Detector

As mentioned earlier, our focus is on the SMS OTP authentication protocol. The function of the login code detector is to analyze a set of apps and generate an *OTP List* of the apps that implement SMS OTP login activities. AUTH-EYE achieves this in two steps: *app decompilation* and *Login Activity locating*, which are detailed as follows.

4.1.1 App Decompilation. AUTH-EYE is built on top of the JEB Android decompiler [42]. The login code detector first takes apps as inputs and uses JEB to decompile them into their java source code.

4.1.2 Login Activity Locating. Since AUTH-EYE only focuses on OTP authentication, it first needs to distinguish Activities of OTP authentication from that of password authentication. However, since both Activities are named similarly in layout files, it is hard for AUTH-EYE to distinguish OTP authentication via the layout files only. Therefore, AUTH-EYE examines the java source code of each app to identify those which implement login activities. AUTH-EYE first looks for *Activities* that were declared by developers (i.e., customized classes and functions) and then identifies functions that fulfill login *Activities*. The examination steps of how AUTH-EYE recognizes OTP login *Activities* in apps are detailed below.

Customized Package Selection. We observe that *Activities* commonly exist in customized packages (i.e., declared by developers) and seldom exist in third-party libraries. Therefore, AUTH-EYE needs to distinguish customized packages from the standard packages declared in third-party libraries. AUTH-EYE adopts a heuristic approach to achieve this target: it ONLY collects class names of Activities in an app and analyzes the first two fields in a class name. For example, in the class name `cx.itxxx.usercenter.activity`[4], the first two fields, `cn.itxxx`, indicate the identity of the developer, while the last part, `usercenter.activity`, refers to the functions that are related to an *Activity*. In one app there often exist many Activities. AUTH-EYE defines the most frequently appeared first-two-fields prefix as the developer's information, and then deems packages with this developer's information as customized packages. According to our manual inspection on 2,210 popular Android apps, 2,153 apps follow this pattern and thus AUTH-EYE could apply this approach to locate customized packages.

Login Function Identification. Once the customized packages are identified, AUTH-EYE locates classes and functions with login *Activities*. A common strategy for understanding a function semantically is to search for specific words and dependency patterns. However, in our case this strategy fails to extract semantic information effectively for the following reasons:

- *Non-Uniform Representation:* Developers often choose function and variable names arbitrarily. They often use different names for functions and variables with the same purpose, such as the `AccountLoginActivity` login functions and `PhoneLogin`. For a matching approach to succeed, we thus need a dataset with a large number of potential keywords.
- *Vague Explanation:* Developers may use identical words but combine these words in different orders to name different functions. The meanings of these functions are significantly different. This makes a simple keyword comparison inaccurate. For example, the function `LoginPhoneAct` refers to the activity of extracting phone settings, but `PhoneLoginAct` specifies login via mobile phone.
- *Unclear Expression:* Developers often use abbreviations and colloquial terms (e.g., `AccountAct`) to declare an *Activity*. Moreover, the decompilation replaces some

[4]We used 'x' to conceal the details of this developer

APIs by using abstract formats, e.g., `UserLogin.a`. Abbreviations and abstract formats like this are more difficult to recognize through a simple keyword matching.

- *Partial Decompilation:* We cannot completely decompile apps with protected code snippets. Hence, the patterns extracted through the data and control dependencies may be inadequate.

To gain a further understanding of these difficulties, we manually inspected the source code of each app. We found that class names are usually fully decompiled, and follow some specific formats. For example, developers use similar words (e.g., `login`, `auth`) with similar formats to name a login *Activity*. Therefore, we propose a natural language processing [33] (NLP) based approach, which is widely used to evaluate the semantic similarity between a pair of words, to extract the semantic information from class and function names and so address the second challenge.

We first manually construct a *reference set* for similarity comparison. We collected the login-related class names and function names from 4,665 repositories posted on Github[5], where open source apps display the code with login *Activities*. A context is required for measuring the semantic similarity between an unknown name and names in the reference set. Because we compare the semantic similarity of two words defined in programming code, text corpora such as Wikipedia and Google Book Ngram Corpus are not appropriate. Instead, we generated our own *code corpus* by using all posts on StackOverflow.To compare an unknown name with names in the reference set, AUTH-EYE converts the words in code corpus into sets of vectors by using Word2vec [34] and computes a cosine distance between the unknown name and each name in the reference set. The cosine distance between two words gives a measure of their semantic similarity, where a greater cosine distance represents a higher semantic similarity. Using the reference set, AUTH-EYE proceeds according to the following steps to identify functions related to login *Activities*:

(1) AUTH-EYE splits each name into several words based on the occurrence of uppercase letters to improve the accuracy of a comparison between an unknown name and names in the reference set. We assume the standard programming practice in which the names of classes and functions are usually a combination of several words, capitalized at word boundaries.
(2) AUTH-EYE compares each word with the names in the reference set and computes the corresponding similarity scores. The highest score found for the word is taken as its similarity score.
(3) AUTH-EYE calculates the semantic similarity by taking the average similarity of all words in the name. If the average similarity score is higher than a *threshold*, the corresponding class name is labeled as a "login".
(4) AUTH-EYE runs steps (1) - (3) iteratively to distinguish "login" classes and then repeats the three steps to label "login" functions in these classes.

For example, consider the function name `doLogin`. The highest semantic similarity score is 0.43 if AUTH-EYE compares the entire name with each word in the reference set. If the name is divided into single words as "`do`" and "`Login`", its average semantic similarity is 0.66 as the similarity scores for two words are 0.33 and 0.98, respectively.

SMS OTP Identification. AUTH-EYE examines the identified login function to determine whether its functionality is SMS OTP login.

Even though the java code for each app may be only partially decompiled, the layout XML file can be fully extracted. Therefore, AUTH-EYE identifies the apps implementing SMS OTP login *Activities* by analyzing the layout XML files instead of analyzing the login function code. For each identified login function, AUTH-EYE uses UI Automator [1] to find the name of the corresponding layout XML file from the "public.xml" file. In the layout XML file, UI Automator parses all the information describing each widget, such as type, text, orientation, and position (i.e., its layout).

Differently from the case with function declarations, we find that each widget is named formally. Thus, we manually collected a set of keywords (e.g., "smscode" and "mobilephone") from 13 repositories posted on Github [9][6] to construct a keyword list. In order to identify whether a widget is related to SMS OTP login, AUTH-EYE chooses the widgets for edit text and button. For each widget, AUTH-EYE compares the text in the field of `android:text` with the words in the keyword list. If any keyword is included in the widget text field, an SMS OTP login is identified, which indicates that its server validates the user's identity through the OTP authentication protocol. AUTH-EYE finally generates an OTP list containing those apps that are identified using SMS OTP login. The widget descriptions, that are relevant to SMS OTP login, are also included.

4.2 Auth Message Analyzer

Taking the OTP app list as an input, the Auth message analyzer of AUTH-EYE then executes OTP login *Activities* through each listed app to interact with its server. By examining server responses and checking the OTP values, AUTH-EYE determines whether the implemented OTP authentication protocol violates any of the OTP rules.

[5]Repositories on Github: https://github.com/search?q=Login+Android

[6]13 repositories on Github: https://github.com/search?q=OTP+Login+Android

4.2.1 OTP Login Execution. AUTH-EYE utilizes Monkey tool, an UI/Application Exerciser [2], which triggers SMS OTP login *Activities* by generating pseudo-random streams of user events (e.g., clicks and button touches). However, the executions of Monkey are somewhat imprecise so that some redundant *Activities* may be performed accidentally, causing unexpected errors. To improve the efficiency and effectiveness of the automated OTP login testing, AUTH-EYE makes use of the orientation information parsed by UI Automator to precisely locate the widgets (i.e., edit text and button) that are related to SMS OTP login *Activities*.

To execute a login *Activity*, AUTH-EYE calls the function `dispatchString()` to enter a valid mobile phone number into the edit text widget. It then clicks the button to send the mobile phone number to the server, requesting a pseudo-random OTP value. The server response is a text message containing the OTP value and some textual descriptions. AUTH-EYE needs to accurately extract the OTP value in this message (note that AUTH-EYE has been given root permission for the tested Android phone in advance to extract the SMS messages from database `/data/data/android.providers.telephony/databases/mmssms.db`). We thus manually analyzed response messages generated by executing the SMS OTP login in 200 apps and found that the messages with OTPs followed formal formats. Therefore, we created a list of keywords that describe OTP values such as "password", "OTP". In detail, AUTH-EYE applies a keyword matching to parse each response message. Given the keywords, AUTH-EYE extracts the OTP value from each message in the following steps:

(1) AUTH-EYE pre-processes a response message by applying Porter Stemmer [32, 41] to convert words to their root forms. For example, the root form of "time" and "times" is "time".
(2) AUTH-EYE divides the message into several *blocks* based on the text spaces. Each block contains a numeric value or a word.
(3) AUTH-EYE searches for the block whose word matches any of the predefined keywords. If a match is found, AUTH-EYE selects the content in the subsequent numeric block as the OTP value.

An interesting observation is that many Android apps nowadays ONLY use OTP authentication and do not adopt a password authentication. In this situation AUTH-EYE could easily conduct the test without considering the password login issue. For those apps with a two-factor authentication (i.e., a login requires both the password and the OTP), AUTH-EYE relies on a manual account registration and login as the prerequisite to conduct the following evaluation.

4.2.2 Evaluating Rule Violations. AUTH-EYE executes the following tests to check each app's compliance with the OTP rules.

R1: OTP Randomness. To assess the randomness of the pseudo-random values generated by a server for each authentication session, AUTH-EYE sends 30 OTP requests to each server and parses the response messages to extract a sequence of OTPs for each server.

AUTH-EYE proceeds in two ways to generate the value sequences for examination. In the first, AUTH-EYE consumes each received OTP before sending a new login request. In the second, AUTH-EYE sends login requests without consuming the values for OTP authentication. From the sequence of OTPs, AUTH-EYE evaluates the randomness from the following two perspectives:

- Repetition: AUTH-EYE identifies whether a subsequence appears periodically in the sequence, or the same value appears repeatedly in the sequence.
- Static: AUTH-EYE examines the sequence with constant values.

We observed some apps reject repeated requests when certain numbers of request are reached, and so prevent acquisition of the 30 values required. In such cases, we wait until the validation works again before re-starting the test. Most apps only block the account for around 10 minutes, and rarely for one hour. Only a few apps block the account for as long as 24 hours.

It is important to mention that if the value sequence passes the above checks, it can only be regarded as potentially random. In this study, we did not test the values in the sequence using more rigorous tests for randomness because that requires a larger number of OTPs. Due to the constraints set in each validation system, collecting a sufficiently large number of OTPs is time-consuming, and even perhaps impossible in practice.

R2: OTP Length. AUTH-EYE checks the length of each OTP, which should be at least six digits. If a server generates an OTP with length less than six digits, AUTH-EYE labels the corresponding app as vulnerable.

R3: Retry Attempts. As the number of allowed attempts is not suggested by RFCs, we consider apps that allow more than five attempts to be insecure. AUTH-EYE first requests a valid OTP from the server to test this property. It then generates a fake OTP by using '0' to replace all the digits in the valid OTP (or using '1' if the valid OTP happen to be all-zeros). The fake OTP value is used for testing the existence of a retry limit. AUTH-EYE then submits the incorrect value five times and analyzes the five responses sent back from servers. Since the error message is not shown as an SMS message,

AUTH-EYE relies on Burp Suite [38] to collect responses from each server and store them in a log file for further parsing.

To identify the limitation on retry attempts, AUTH-EYE compares the five error messages without considering the values in the messages. If the five messages are identical, it implies that the validation system may not limit the number of attempts. In that case AUTH-EYE sends the fake value repeatedly to confirm whether there is any limit. AUTH-EYE terminates this procedure under two circumstances: 1) a different message is received, such as "Too many errors", or 2) AUTH-EYE has made 20 attempts[7]. The first circumstance implies that there is a limitation, and the number of attempts so far performed by AUTH-EYE is the maximum allowed attempts. The second case indicates that the validation system may allow unlimited attempts.

If the five error messages are not the same, AUTH-EYE then identifies the word describing attempts from these messages. AUTH-EYE searches for the format as a value followed by the word "time", which refers to how many more attempts may be made. Setting a time delay is a possible additional protection mechanism for retry attempts. Given the five error messages, AUTH-EYE first searches for the word "delay" and then finds the format as a value followed by a time-related word (i.e., "second/s", "minute/min", or "hour/h"). The value found is extracted as the required delay before another attempt.

R4: OTP Consumption. To identify whether the validation system of OTP authentication accepts a re-used OTP, AUTH-EYE first requests and consumes a valid OTP. It then attempts the consumed OTP again. If the validation succeeds for the second time, it indicates that the implemented OTP authentication does not check or remember the provided OTP for each authentication session, and permits repeated use of OTPs across multiple sessions.

R5: OTP Expiration. Given response messages, AUTH-EYE searches for the word "expire" and extracts the value after this word, that is, the validation time of the received OTP. To measure the expiration interval, we set a timer in AUTH-EYE. Once the OTP message is received, AUTH-EYE starts the timer and repeatedly sends the OTP to the server for validation until it expires. If AUTH-EYE can be validated successfully, a violation is detected (i.e., the server does not check the expiration of the OTP adequately).

R6: OTP Renewal Interval. To identify apps that violate this rule, AUTH-EYE provides a valid OTP at times corresponding to different time intervals. Because the RFC recommends 30 sec as the optimal time interval, we set the time intervals to [0, 30s], [30s, 60s], [60s, ∞]. In our test, AUTH-EYE first uploads the OTP immediately after it was received (i.e., within 30 seconds). Then, AUTH-EYE requests a new OTP and submits the value within 30 to 60 seconds. If the request succeeds, AUTH-EYE asks for another OTP value and resubmits it after 60 seconds. AUTH-EYE repeats this test with the renewal intervals of [1min, 5min], [5min, 10min], [10min, 30min], [30min, 60min], [60min, 24h], and [24h, ∞] until it is rejected by the server (or finds a still available OTP after 24 hours). If a server accepts an OTP with a lifetime more than 30 seconds, AUTH-EYE considers it as an insecure one.

[7]We choose 20 attempts after considering the potential legal issue in mainland China. Also, we can easily add the guess times to determine which apps are actually vulnerable.

5 EVALUATION

Our evaluation has two goals. The first is to assess the effectiveness of AUTH-EYE in automatically analyzing the implementations of OTP authentication protocols in Android apps and verify that their implementations comply with the OTP rules. The second is to use AUTH-EYE to gain insights into the frequency of violations of OTP rules in real-world Android apps.

5.1 Dataset

We built our app dataset by downloading 3,303 top list apps from both Google Play and Tencent MyApp markets (986 from Google Play and 2,317 from Tencent) between February and April 2019. The dataset contains apps in 21 categories including *Beauty*, *Books & Reference*, *Communication*, *Education*, *Entertainment*, *Finance*, *Health & Fitness*, *Lifestyle*, *Map & Navigation*, *Medical*, *Music & Audio*, *News & Magazine*, *Parenting*, *Personalization*, *Photography*, *Productivity*, *Shopping*, *Social*, *Tool*, *Travel & Local*, *Video Players & Editors*. We selected from each category the *recommended apps* (about 150 apps in each category, and the most active one has around 3 billion downloads).

We observed that many apps also provide the option of login via a third party (e.g., OAuth). Note that in this paper we only assess apps with customized OTP authentication protocols, and those which use third-party authentication services with open-authentication are out of the scope of this paper.

5.2 OTP Login Activity Recognition

The first task of AUTH-EYE is to create an OTP list, i.e., a list of apps implementing SMS OTP. Among the the 3,303 apps in our dataset, AUTH-EYE is able to analyzed 1,364 apps, while other apps adopt app protection measures (e.g., code packing and code obfuscation) to hinder the decompilation and code analysis of AUTH-EYE. We manually inspected the apps that AUTH-EYE failed to analyze to gain some insights:

Table 1: Top-10 login activity names in apps

Login Activity Names	# of apps
Login	105
LoginSuccess	53
doLogin	37
smsLogin	18
onLoginSuccess	16
startLogin	14
requestLogin	14
startLoginActivity	13
supportSmsLogin	13
serverBindLoginRequest	13

- 648 apps are protected using code packing against decompilation, in which their ".class" files are encrypted. These files will only be decrypted during app execution. Since we cannot extract the source code from encrypted apps, AUTH-EYE is unable to locate their login Activities and cannot execute them.
- AUTH-EYE are not able to analyze 1291 apps because 1) 695 of them use code obfuscation to prevent the code from being analyzed and 2) 596 apps are unable to be executed due to potential anti-debugging code.

We argue that AUTH-EYE could also adopt advanced analysis technique such as unpacking to handle these issues, but this often involves manual efforts (e.g., patching anti-debugging code) and is not scalable. More importantly, we observe that apps developed by large companies (e.g., Microsoft, Alibaba, Tencent, Baidu) seldom adopt code protection due to stability and compatibility requirements. Therefore we leave the analysis of protected apps as a future work and only focus on those unprotected apps.

AUTH-EYE identified 1069 (78.3%) with declared login Activities in successfully analyzed 1,364 apps, and the top-10 commonly used login *Activity* names are listed in Table 1. It is clear from the list that developers do prefer to use the word "login" to describe a login *Activity*. Given the list of apps with identified login Activities, AUTH-EYE then further identified how many implement OTP authentication. In total, 544 (58.2%) app adopt OTP authentication. Among these 544 apps, 354 use two-factor authentication (both password authentication and OTP authentication), while 190 apps only contain OTP authentication. In this study, we only discuss the validation OTP authentication and leave the evaluation of password authentication protocols as future work. Hence, our discussion focuses only on the apps in the OTP list, i.e., 544 apps implementing SMS OTP authentication. Note that for apps with password authentication involved, we manually registered an accounts in those apps and typed in the combination of username and password.

5.3 Results

5.3.1 Rules Violations. Table 2 lists the number of apps that violate the OTP rules (see Section 2). Only eight apps out of the 544 apps did not violate any of the OTP rules. We now discuss the detected violations of OTP rules in the order of their prevalence.

Table 2: Violations of OTP rules

OTP Rules	# of apps
R6: OTP Renewal Interval	536
R3: Retry Attempts	324
R2: OTP Length	209
R4: OTP Consumption	106
R1: OTP Randomness	71
R5: OTP Expiration	40

R6: OTP Renewal Interval. A large number of apps, 536 in total, violated this rule, making it the most frequently violated OTP rule. Only eight apps follow the requirement proposed by **R6**. Further inspection revealed that in 165 apps, the OTP validation systems did not require OTP values to be renewed. For the remaining failed apps (i.e., 371 apps), the intervals to renew OTP values set by their validation systems are shown in Figure 2. Most validation systems (122 apps) are set to renew OTP values at intervals between 5 minutes to 10 minutes. The validation systems of 112 apps generate new OTP values within the time interval of one minute to five minutes. Even worse, AUTH-EYE identified that the validation systems in 15 apps accept OTPs that have been delayed for 24 hours. This design results in the TOTP authentication protocol behaving no better than a normal OTP authentication protocol. The developers of these apps might deliberately choose this option since accepting a large range of delays as valid is much more user-friendly.

R3: Retry Attempts. This rule limits the number of retry attempts allowed by validation systems. It is the second most violated OTP rule. AUTH-EYE identified 324 (59.6%) apps out of 544 apps violating this rule, i.e., allowing more than five attempts.

Figure 3 shows the number of attempts allowed by validation systems. Only 220 (40.44%) apps have OTP validation systems complying with the rule, and most of these apps (77.2%) are from the category of Shopping and a few are from the Social category.

For the other apps that violate **R3** (i.e., 324 apps), 111 apps allow 6 to 10 retry attempts, and 31 apps allow 11 to

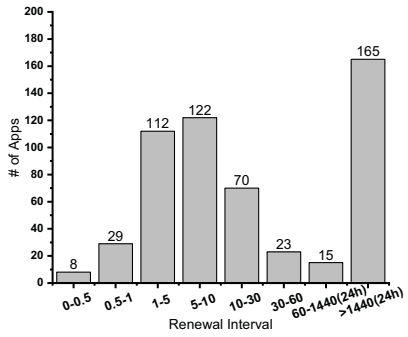

Figure 2: OTP renewal interval (Minutes)

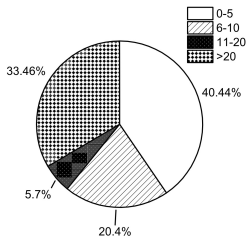

Figure 3: Number of retry attempts allowed in apps

20 attempts. Because AUTH-EYE was set to send a fake OTP at most 20 times for each app, it identified that the OTP validation systems in 126 apps still work after 20 times of retry. We surmised that these validation systems may not implement any limitation and are thus vulnerable to brute force attacks.

Additionally, AUTH-EYE identified the delay protection implemented in the OTP validation systems of 97 apps. In these apps, the user has to wait for a specific period if an incorrect OTP value is entered. The waiting period set in these apps is usually one minute.

R2: OTP Length. The third most violated OTP rule is setting the length of OTP values at fewer than six digits. As mentioned in Section 2, generating an OTP with short length (i.e., length < 6) negates the security benefits of the OTP authentication protocol.

In total, the validation systems in 209 apps use values with less than six digits as OTP values. Although the OTP length could be set at 10 digits, we discovered that all the validation systems generate OTPs with at most six digits.

R4: OTP Consumption. This rule is violated by 106 apps out of the 544 apps. Here, users are allowed to reuse an OTP for identity verification. A unique value for each validation session is essential in the OTP authentication protocol to protect against replay attacks. Accepting a repeated OTP value negates the benefit of using an OTP and can even make the OTP authentication protocol weaker than a password authentication protocol.

Apps violated this OTP rule are only from eight categories, Shopping, Video Players & Editor, Books & Reference, Music & Audio, Travel & Local, Entertainment and productivity. 37.7% vulnerable apps and 18.9% vulnerable apps are from the categories of Books & Reference and Video Players & Editor, respectively.

R1: OTP Randomness. This rule was violated by 71 apps. two types of errors are identified by AUTH-EYE: repeated values and static values.

AUTH-EYE found 56 apps generating repeated OTP values. To be specific, 21 apps generate a sequence of unique OTP values and then repeat the same sequence. The validation systems of 35 apps use the same OTP for n different requests, that is, the same OTP value is repeated n times. Based on our manual inspection, each value is repeated two or three times (i.e., $n = 2$ or $n = 3$).

In addition, AUTH-EYE discovered that 15 apps provide only static OTP values to users for OTP authentication. This type of error makes the OTP authentication protocol perform as a simple password authentication protocol, in which the username is the user's mobile phone number and the password is the fixed (short) OTP value. An attacker can then easily access the user's account if the mobile phone number is leaked because the OTP value is shorter and simpler than a static password set by the client.

The above results indicate that developers might not be aware of the critical importance played by randomness in authentication.

R5: OTP Expiration. AUTH-EYE identified 73 apps that use TOTP authentication protocols, in which the OTP value may expire. Interestingly, only 33 apps reject the OTP value if it is expired, while the identity verification of 40 apps passed by providing expired OTP values. This implies that the implemented TOTP authentication protocols fail to work properly in the corresponding servers.

For the remaining 471 apps, AUTH-EYE did not discover any expiration set for OTP values by only analyzing the responses. We might suppose that their validation systems allow the OTP values to be valid forever.

We manually inspected those 1069 apps and found that 934 (87.4%) of them did implement login Activities. AUTH-EYE mistakenly identified some apps because their *Activity* names, such as `loginFail` and `thirdLogin`, have higher semantic similarity scores.

Table 3: Violation of multiple OTP rules

# of apps	multiple-rules violated
65	**R2** & **R4**
13	**R1** & (**R2** or **R3**)
9	**R4** & **R5**
2	**R2** & **R3**

5.3.2 Results for Multiple-Rules Violations. We also investigated the status of multiple OTP rules violation in our tested apps, and listed the cases that severely threat to the security of OTP authentication in Table 3. As shown in the table, the most frequently occurred situation is the violation of both **R2** and **R4** (65 apps). In this situation, an attacker could guess the OTP through a brute force attack, since the number of legal OTPs is limited and the server also accept a used OTP. Another common mistake is that 13 apps not only used an immutable OTP, but also violate other rules such as allowing an attack to guess the OTP, or always reuse the immutable OTP if a legal user does not enforce a new login request. There are also nine apps violate both **R4** and **R5**, and two apps violate **R2** and **R3** simultaneously. All those apps that violate multiple OTP rules are considered as highly vulnerable, and we have contacted the developers and reported these issues.

Note that we found all "potential vulnerable" apps violated **R6** (i.e., the valid time window exceeds 30 seconds). Compared with a single rule violation, violations of both **R6** and other rules will increase the risk. However, the violation of **R6** often does not directly lead to an attack. Therefore, we do not consider this issue in our multiple rules violation investigation.

5.4 Case Studies

This section aims to highlight insights from case studies based on our manual inspections.

Matchless Functionality. We found some apps whose validation systems do not match with the responses.

–*Expiration.* We investigated a game manager app with more than 100,000 downloads. It transmits messages in secure ciphertext formats and the server responses suggest that authentication protocols (i.e., password authentication and OTP authentication) are correctly implemented. However, AUTH-EYE flagged this app because its validation system still accepts "expired" OTPs. By parsing the server responses, AUTH-EYE discovered that each OTP expired after 30 seconds. However, when AUTH-EYE consumed each OTP after 1 minute and 10 minutes, it passed the validation. We agree that remembering and validating OTP values for all login requests consume a large amount of storage and memory on the server side; but app security makes it essential to implement an efficient reset method for clearing expired values.

–*Consumption.* Users commonly use finance apps to manage their investments. It is crucial that financial information is protected at all times. However, we found that the validation systems in three financial apps accept previously consumed OTPs. The OTP authentication of a financial app was labeled by AUTH-EYE as vulnerable. By checking its responses, we found that this app violates almost all OTP rules except for **R1** (OTP Randomness) and **R5** (Retry Attempts). The only protection scheme implemented is that its validation system blocks the user's account and does not generate any OTP values if the user keeps sending requests more than five times. While verifying user identity, this app not only accepts consumed OTP values, but also transmits mobile phone numbers and OTP values in plaintext. This means that users' private information (i.e., login information and private data) is exposed to attackers.

Deceptive Randomness. Apps violating **R1** (OTP Randomness) are from the categories of Beauty, Finance, News & Magazine, Photography and Video Players & Editors. The percentage from each category with violations is 11%, 7%, 33%, 3%, and 46%, respectively. We investigated these apps in details. For 6 of the 15 apps which generate static values for OTP authentication, the OTP value is only renewed when the previous one is consumed. The other apps keep sending the same value to users.

Exposed Transmission. As well as analyzing server responses, AUTH-EYE monitored traffic messages to identify whether an error occurred. From traffic messages, AUTH-EYE identified that most messages containing OTP values are not well-protected. The validation systems of 188 apps transmit the OTP values in plaintext over the unsecured network. 36 apps protect OTP values by using only an MD5 hash without salt, which is considered insecure [19]. With respect to message transmission, we found that the most secure category is Travel & Local, where 98.7% apps encrypt their transmitted messages. The categories of Shopping and Social perform the worst with only 72% and 74.6% apps being secure. We observed that a Music app (10,070,000 downloads) only uses the user's mobile phone number as the password no matter what user password and OTP are provided, and only the mobile phone number is transmitted to the validation system.

5.5 Discussion

We have demonstrated through an experimental evaluation that AUTH-EYE is effective in assessing the design and implementation of OTP authentication protocols in Android apps. However, it has some drawbacks, outlined as follows.

- **Discovering Vulnerabilities.** AUTH-EYE executes each app for SMS OTP validation and points out OTP rules violated by the validation system. However, it is difficult for AUTH-EYE to discover what vulnerabilities exist in the implementation and where they are located Consider a case that violates **R1** (i.e., OTP randomness). The most popular way of implementing this in Android is to invoke the function `SecureRandom(.)`, a pseudo-random number generator. In practice, it should not be seeded with a constant number; otherwise, the function will produce a predictable output sequence across all implementations. However, some developers still use seeds as "000000" or "123456" [31]. Such details cannot be inferred in general by only analyzing the server responses.
- **Vulnerability Certainty.** Currently, all the implementations that violate any of the OTP rules are tagged as vulnerable. There might be other protection mechanisms implemented in the validation system, beyond those known to AUTH-EYE. For example, we discovered that some Finance apps transmitted mobile phone numbers and OTPs in ciphertext or over a secure connect. These protection schemes confirm that transmissions are under secure circumstances, as long as the cryptographic primitives or secure connection are correctly implemented. This is, however, outside the scope of this work.
- **Black Box Analysis.** AUTH-EYE treats the validation system as a black box, and only analyzes server responses. We assume that such responses reflect functionalities implemented in the code. However, this assumption does not always hold as our manual inspections described in Section 5 found, such as in the case of the Finance app discussed in Section 5.4. Based on a given response, one may conclude that the implementation complies with the implementation rules, but nevertheless functionalities defined in the validation system may actually not be correctly implemented.
- **Field Identification.** AUTH-EYE identifies the required information from the server responses through keywords match only. Nonetheless, the formats of the responses are not always shown as the same. AUTH-EYE might miss some responses in other format or identify an incorrect information from the responses.

6 RELATED WORK

This section provides a brief review of related work.

6.1 Security Analysis of One-Time Password Authentication Protocols

Several vulnerability detection approaches and protection schemes are proposed to defend against attacks on the steps of protocols, e.g. transmission and password generation. TrustOTP [44] builds a TrustZone to protect against attacks such as Denial-of-Service (DoS). It isolates OTP code and data from the mobile OS to ensure that the generated OTPs and seeds are secure even if the mobile OS is compromised. Differently, Hamdare *et al.* [23] and Eldefrawy *et al.* [16] focused on securing the authentication during OTP transmission and OTP generation, respectively. Hamdare *et al.* proposed a scheme to protect the OTP mechanism used for e-commerce transactions from Man-In-The-Middle (MITM) attacks. Instead of transmitting a simple OTP, they combined the OTP with a secure key and created a new transaction password by encrypting the combination using RSA. Eldefrawy *et al.* focused on securing OTP generation by generating multiple OTPs for both service providers and consumers. The low computation cost of their scheme makes it suitable for mobile devices.

The focus of previous work is to enhance the security of OTP authentication protocols. However, we focused on analyzing the correctness of OTP implementations and proposed an approach to check server responses instead of directly performing code analysis. Mulliner *et al.*. [37] proposed a similar research that conducted a survey by introducing several attacks and weaknesses of SMS OTPs such as SIM swap attack, wireless interception. To protect SMS OTP attacks, the possible defense techniques are generally provided. However, details to exploit vulnerabilities and protect SMS OTP authentication are not given. Dmitrienko *et al.* [12] investigated the implementation of two-factor authentication of well-known Internet service providers (e.g., Google, Dropbox, Twitter, and Facebook). They applied cross-platform (i.e., PC-mobile) infection to control both PC and mobile devices involved in the authentication protocol and identified weaknesses that could be exploited in the SMS-based transaction authentication schemes of four large banks and Google. Their approach determines whether an authentication implementation is secure, but cannot provide detailed information on the causes. AUTH-EYE, however, does identify the causes of implementation flaws.

6.2 Dynamic Vulnerability Analysis of Mobile Apps

Several approaches have been proposed for dynamic analysis of mobile apps. The approach by Zuo *et al.* [53] detects vulnerabilities arising from SSL error handling in the mobile platform. They statically identified customized error handling processes rewritten by developers. For each process,

their system executes an event to trigger the error and checks whether the error is processed correctly. This approach requires access to the source code. In contrast, AUTH-EYE assumes that the server source code is not available, and uses only decompiled Android app code.

D'Orazio *et al.* [13] relied on an adversary model specifying secure and insecure states to detect vulnerabilities that can expose users' sensitive data in mobile devices. The drawback of their approach is that complete code coverage cannot be ensured through app execution. AUTH-EYE instead achieves full code coverage because the entire login activities in apps are examined; thus, all the authentication relevant functionalities can all be triggered. Also relying on an attack scenario, IntentDroid [24] identifies Inter-Application Communication (IAC) vulnerabilities by executing attacks on eight specified vulnerabilities. It analyzes activity components of apps by implementing attacks based on effect path coverage, with low overhead. Although IntentDroid only uses a small set of tests to achieve high coverage analyses, path analysis mechanisms cannot be accurately applied to decompiled app code.

AUTH-EYE is similar to SmartGen [52], which performs in-context analysis to expose harmful URLs through symbolic execution in mobile apps. Since server URLs are invisible, SmartGen triggers the appropriate execution through APIs in app code. The main difference between SmartGen and AUTH-EYE is, however, that AUTH-EYE does not rely on dependency patterns to identify the target functions. It uses only limited code information (i.e., class names and function names) that are extracted easily and completely. SmartGen targets on detecting of hidden malicious URLs whereas AUTH-EYE identifies violations of the OTP rules in implementations of OTP authentication protocols.

7 CONCLUSION

In this paper we defined six OTP rules based on the relevant RFCs and proposed a novel automated system, AUTH-EYE, to check for violations of those rules. We used AUTH-EYE to perform an empirical study on a large number of Android applications. Our approach treats each server as a black box and infers the correctness of its OTP implementation code by analyzing server responses generated after a sequence of login requests. We assessed 3,303 Android apps and identified 544 apps implemented OTP authentication. Only eight of these 544 apps correctly implemented the OTP authentication protocols (i.e., satisfied all six OTP rules). Further analysis revealed the surprising fact that the validation systems of apps in security-critical domains, such as Finance, Shopping, and Social, are not as secure as one might expect. The example cases discussed in Section 5.4 show that poor implementations make users' accounts vulnerable to attack and may even expose private data directly. As future work, we plan to extend AUTH-EYE to examine additional OTP rules and perform a more extensive survey of real-world app OTP security. A new dynamic analysis tool, CuriousDroid [10], is introduced. It is a context-based technique and can achieve higher accuracy than Monkey. We will refer to this technique to improve AUTH-EYE. The interested readers could also access our Github page[8] to obtain source code of AUTH-EYE and help improve the analysis.

ACKNOWLEDGEMENT

This work was mainly supported by CSIRO Research Office. It was also partially supported by the General Program of National Natural Science Foundation of China (Grant No.61872237), the Key Program of National Natural Science Foundation of China (Grant No.U1636217), and the Major Project of Ministry of Industry and Information Technology of China ([2018] No.36).

REFERENCES

[1] Android [n. d.]. UI Automator. https://developer.android.com/training/testing/ui-automator.

[2] Android. [n. d.]. UI/Application Exerciser Monkey Tool. https://developer.android.com/studio/test/monkey.

[3] Paul Ashley, Heather Hinton, and Mark Vandenwauver. 2001. Wired versus wireless security: The Internet, WAP and iMode for e-commerce. In *In Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*. IEEE, 296–306.

[4] Zhongjie Ba and Kui Ren. 2017. Addressing smartphone-based multi-factor authentication via hardware-rooted technologies. In *In Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1910–1914.

[5] CITI Bank. [n. d.]. CITI Bank Mobile Token. https://www.citibank.com.au/aus/banking/citi-mobile-token.htm.

[6] Antonio Bianchi, Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2017. Exploitation and mitigation of authentication schemes based on device-public information. In *In Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*. ACM, 16–27.

[7] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. 2012. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *In Proceedings of the 33rd IEEE Symposium on Security and Privacy (S & P)*. IEEE, 553–567.

[8] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *In Proceedings of the 22nd Usenix Security Symposium (USENIX)*. 131–146.

[9] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Mining java class naming conventions. In *In the proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 93–102.

[10] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2016. CuriousDroid: automated user interface interaction for android application analysis sandboxes. In *International*

[8] https://github.com/GoSSIP-SJTU/auth-eye

Conference on Financial Cryptography and Data Security. Springer, 231–249.

[11] Joseph A Cazier and B Dawn Medlin. 2006. Password security: An empirical investigation into e-commerce passwords and their crack times. *The Journal of Information Systems Security* 15, 6 (2006), 45–55.

[12] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. 2014. On the (in) security of mobile two-factor authentication. In *In Proceedings of the 18th International Conference on Financial Cryptography and Data Security (FC)*. Springer, 365–383.

[13] Christian J D'Orazio, Rongxing Lu, Kim-Kwang Raymond Choo, and Athanasios V Vasilakos. 2017. A Markov adversary model to detect vulnerable iOS devices and vulnerabilities in iOS apps. *The Journal of Applied Mathematics and Computation* 293 (2017), 523–544.

[14] D Eastlake 3rd, Steve Crocker, and Jeff Schiller. 1994. *Randomness recommendations for security*. Technical Report.

[15] D Eastlake 3rd, J Schiller, and Steve Crocker. 2005. *Randomness requirements for security*. Technical Report.

[16] Mohamed Hamdy Eldefrawy, Khaled Alghathbar, and Muhammad Khurram Khan. 2011. OTP-based two-factor authentication using mobile phones. In *In Proceedings of the 8th International Conference on Information Technology: New Generations (ITNG)*. IEEE, 327–331.

[17] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. 2018. Android malware familial classification and representative sample selection via frequent subgraph analysis. *The Journal of IEEE Transactions on Information Forensics and Security (TIFS)* 13, 8 (2018), 1890–1905.

[18] John Franks, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart. 1999. *HTTP authentication: Basic and digest access authentication*. Technical Report.

[19] Praveen Gauravaram. 2012. Security Analysis of salt|| password Hashes. In *In Proceedings of the 1st International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*. IEEE, 25–30.

[20] Google. [n. d.]. Google Authenticator. https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2&hl=en_AU.

[21] Nancie Gunson, Diarmid Marshall, Hazel Morton, and Mervyn Jack. 2011. User perceptions of security and usability of single-factor and two-factor authentication in automated telephone banking. *The International Journal of Computers & Security* 30, 4 (2011), 208–220.

[22] Neil Haller, Craig Metz, Phil Nesser, and Mike Straw. 1998. *A one-time password system*. Technical Report.

[23] Safa Hamdare, Varsha Nagpurkar, and Jayashri Mittal. 2014. Securing SMS based one time password technique from Man in the middle attack. *arXiv preprint arXiv:1405.4828* (2014).

[24] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic detection of inter-application communication vulnerabilities in Android. In *In Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 118–128.

[25] Kyle Ingols, Richard Lippmann, and Keith Piwowarski. 2006. Practical attack graph generation for network defense. In *In Proceedings of the 22nd IEEE Annual Computer Security Applications Conference (ACSAC)*. IEEE, 121–130.

[26] Jongpil Jeong, Min Young Chung, and Hyunseung Choo. 2008. Integrated OTP-based user authentication scheme using smart cards in home networks. In *In Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*. IEEE, 294–294.

[27] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 2014. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *In Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 66–77.

[28] Pawel Laka and Wojciech Mazurczyk. 2018. User perspective and security of a new mobile authentication method. *The Journal of Telecommunication Systems* 69, 3 (2018), 365–379.

[29] Leslie Lamport. 1981. Password authentication with insecure communication. *The Journal of Communications of the ACM* 24, 11 (1981), 770–772.

[30] Jaeho Lee, Ang Chen, and Dan S Wallach. 2019. Total Recall: Persistence of Passwords in Android.. In *In Proceedings of The Network and Distributed System Security Symposium (NDSS)*.

[31] Siqi Ma, David Lo, Teng Li, and Robert H Deng. 2016. Cdrep: Automatic repair of cryptographic misuses in android applications. In *In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 711–722.

[32] Siqi Ma, Shaowei Wang, David Lo, Robert Huijie Deng, and Cong Sun. 2015. Active semi-supervised approach for checking app behavior against its description. In *In Proceedings of the 39th IEEE Annual Computer Software and Applications Conference (ICSAC)*, Vol. 2. IEEE, 179–184.

[33] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT press.

[34] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[35] David M'Raihi, Mihir Bellare, Frank Hoornaert, David Naccache, and Ohad Ranen. 2005. *Hotp: An hmac-based one-time password algorithm*. Technical Report.

[36] David M'Raihi, Salah Machani, Mingliang Pei, and Johan Rydell. 2011. *Totp: Time-based one-time password algorithm*. Technical Report.

[37] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. 2013. SMS-based one-time passwords: attacks and defense. In *In Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 150–159.

[38] PortSwigger. [n. d.]. Burp Suite. https://portswigger.net/burp.

[39] William K Pratt, Julius Kane, and Harry C Andrews. 1969. Hadamard transform image coding. *In Proceedings of the IEEE Journals and Magazines* 57, 1 (1969), 58–68.

[40] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C Mitchell. 2005. Stronger Password Authentication Using Browser Extensions.. In *In Proceedings of the 14th Usenix Security Symposium (USENIX)*. Baltimore, MD, USA, 17–32.

[41] SnowBall. [n. d.]. Porter Stemmer. http://tartarus.org/martin/PorterStemmer/java.txt.

[42] PNF Software. [n. d.]. JEB Decompiler. https://www.pnfsoftware.com/.

[43] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. 2017. Large-scale analysis & detection of authentication cross-site request forgeries. In *In Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 350–365.

[44] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming smartphones into secure one-time password tokens. In *In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 976–988.

[45] Mariano Luis T Uymatiao and William Emmanuel S Yu. 2014. Time-based OTP authentication via secure tunnel (TOAST): A mobile TOTP scheme using TLS seed exchange and encrypted offline keystore. In *In Proceedings of the 4th IEEE International Conference on Information Science and Technology (ICIST)*. IEEE, 225–229.

[46] Ignacio Velásquez, Angélica Caro, and Alfonso Rodríguez. 2018. Authentication schemes and methods: A systematic literature review. *The International Journal of Information and Software Technology* 94 (2018), 30–37.

[47] Dong Wang, Xiaosong Zhang, Jiang Ming, Ting Chen, Chao Wang, and Weina Niu. 2018. Resetting Your Password Is Vulnerable: A Security

Study of Common SMS-Based Authentication in IoT Device. *The Journal of Wireless Communications and Mobile Computing* 2018 (2018).

[48] 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 *In Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*. ACM, 61–70.

[49] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *In Proceedings of the 31st IEEE Symposium on Security and Privacy (S & P)*. IEEE, 497–512.

[50] R Winsniewski. 2012. Android–apktool: A tool for reverse engineering android apk files.

[51] Changsok Yoo, Byung-Tak Kang, and Huy Kang Kim. 2015. Case study of the vulnerability of OTP implemented in internet banking systems of South Korea. *The Journal of Multimedia Tools and Applications* 74, 10 (2015), 3289–3303.

[52] Chaoshun Zuo and Zhiqiang Lin. 2017. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *In Proceedings of the 26th International Conference on World Wide Web (WWW)*. International World Wide Web Conferences Steering Committee, 867–876.

[53] Chaoshun Zuo, Jianliang Wu, and Shanqing Guo. 2015. Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps. In *In Proceedings of the 10th ACM on Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 591–596.