Fine with "1234"? An Analysis of SMS One-Time Password Randomness in Android Apps

Siqi Ma[1], Juanru Li[*2], Hyoungshick Kim[3], Elisa Bertino[4], Surya Nepal[5], Diethelm Ostry[5], and Cong Sun[6]
[1] The University of Queensland, slivia.ma@uq.edu.au
[2] Shanghai Jiao Tong University, jarod@sjtu.edu.cn
[3] Sungkyunwan University, hyoung@skku.edu
[4] Purdue University, bertino@purdue.edu
[5] Data61 CSIRO, {surya.nepal, diet.ostry}@data61.csiro.au
[6] Xidian University, suncong@xidian.edu.cn

***Abstract*—A fundamental premise of SMS One-Time Password (OTP) is that the used pseudo-random numbers (PRNs) are uniquely unpredictable for each login session. Hence, the process of generating PRNs is the most critical step in the OTP authentication. An improper implementation of the pseudo-random number generator (PRNG) will result in predictable or even static OTP values, making them vulnerable to potential attacks. In this paper, we present a vulnerability study against PRNGs implemented for Android apps. A key challenge is that PRNGs are typically implemented on the server-side, and thus the source code is not accessible. To resolve this issue, we build an analysis tool, `OTP-Lint`, to assess implementations of the PRNGs in an automated manner without the source code requirement. Through reverse engineering, `OTP-Lint` identifies the apps using SMS OTP and triggers each app's login functionality to retrieve OTP values. It further assesses the randomness of the OTP values to identify vulnerable PRNGs. By analyzing 6,431 commercially used Android apps downloaded from Google Play and Tencent Myapp, `OTP-Lint` identified 399 vulnerable apps that generate predictable OTP values. Even worse, 194 vulnerable apps use the OTP authentication alone without any additional security mechanisms, leading to insecure authentication against guessing attacks and replay attacks.**

***Index Terms*—OTP Authentication Protocol; Mobile Application Security; Pseudo-Random Number Generator; Vulnerability Detection; Randomness Evaluation**

I. Introduction

SMS One-Time Password (OTP) is widely used for authentication and authorization in Android apps [1], which employs a uniquely generated pseudo-random number (PRN) for each login session to verify each user's identity. A pseudo-random number generator (PRNG) is commonly used to generate unpredictable OTP values. Some cryptographically insecure randomness algorithms, such as Mersenne Twister (MT) [2] and Linear Congruential Generator (LCG) [3], have been used in practice. A PRNG using any of these insecure randomness algorithms would generate highly predictable OTP values. Even though the utilized randomness algorithm is secure, the generated PRNs may still be problematic if the algorithm is not implemented correctly (e.g., seeding the randomness algorithm by using a constant) [4], [5].

Many studies [6], [7] have been proposed to analyze the security of pseudo-random number generating algorithms; however, these studies seldom analyze PRNG implementations in apps. The techniques proposed for assessing the PRNG implementations mainly focus on open-source systems (e.g., Linux [8], OpenSSL [9]). However, these techniques rely on code analysis and thus cannot be applied to analyze PRNGs of Android apps. This paper focuses on the following two goals: 1) exploring security vulnerabilities in SMS OTP values generated by Android apps; 2) gaining insights into potential implementation issues of PRNGs used by these vulnerable apps, without accessing the source code of the PRNGs.

Towards fulfilling our goals, we first study the algorithms and functions for generating PRNs to understand what types of vulnerabilities may occur in PRNGs. Next, based on the official RFC documents [10], [11], [12], [13] and research work [14], [15], [16], we introduce three critical randomness rules: **Rule 1** – Do not use a static OTP value; **Rule 2** – Do not generate OTP values according to specific patterns; **Rule 3** – Do not use a constant or predictable seed to initialize a randomness function (Section III). If the OTP values generated for user authentication violate any of the randomness rules, those OTP values can be predicted, and thus the authentication scheme can eventually be cracked.

We develop a novel analysis tool, `OTP-Lint`, to assess the randomness of OTP values and analyze the potential implementation vulnerabilities of the corresponding PRNGs without having access to the PRNG source code. `OTP-Lint` first identifies the login *Activity* declared in each app using a fuzzing-inspired approach. It then recognizes those apps which use SMS OTP authentication through keyword matching. By locating the OTP login widgets, `OTP-Lint` triggers the relevant app functionalities and sends OTP requests to the app server to retrieve OTP values. Finally, `OTP-Lint` evaluates whether the gathered OTP values violate any of the three introduced randomness rules. A major challenge in the vulnerability analysis of OTP values is to determine which algorithm and function are used to generate PRNs and which parameters they are given as input. To address this challenge, we collected PRNG sample codes written in diverse programming languages shared by app developers on GitHub [17] and Stack Overflow [18] to learn the popular

approaches for implementing PRNGs.

We used `OTP-Lint` to analyze 6,431 real-world Android apps, downloaded from both Google Play and Tencent MyApp markets (1,000 from Google Play and 5,431 from Tencent MyApp). Out of these apps, `OTP-Lint` successfully detected that 2,022 apps implemented SMS OTP login, and 399 (19.7%) of these violated the defined randomness rules. Our results demonstrate that a significant number of Android apps would be at real risk of cyber-attacks exploiting such OTP login functions.

We believe that `OTP-Lint` would help service providers and users test whether the OTP authentication in Android apps is securely implemented and highlight potential security issues without accessing PRNG source code. Our main **contributions** are as follows:

- Through an examination of the official RFC documents and research works, we provide insights into potential implementation issues in vulnerable PRNGs and propose three types of randomness rules that must be followed for implementing secure PRNGs.
- To the best of our knowledge, this is the first randomness study of OTP values generated by PRNGs used in Android apps. Without knowing the detailed implementations of PRNGs, we infer the potential vulnerabilities that might exist in the PRNGs through OTP value analysis.
- We build a novel analysis tool, `OTP-Lint`. By triggering OTP authentication in apps, `OTP-Lint` simulates the most common vulnerable PRNG implementations for OTP randomness analysis and checks whether the generated OTP values are vulnerable.
- We used `OTP-Lint` to analyze 6,431 Android apps and detected 399 apps that produce predictable OTP values. Interestingly, 194 vulnerable apps use the OTP authentication alone without any additional security mechanisms, leading to insecure authentication against guessing attacks and replay attacks.

II. Analysis of Randomness Weaknesses

We discuss the widely used randomness algorithms and present the randomness functions in programming languages that can be used for Android apps.

A. Randomness Algorithms

Linear Congruential Generator (LCG). LCG is one of the most popularly used algorithms that generate a sequence of pseudo-random numbers (PRNs), using a discontinuous linear equation. LCG is defined by the recurrence relation $S_{n+1} = a \times S_n + C \mod m$. Starting with a seed, LCG repeatedly applies the recurrence relation to generate the subsequent PRNs. Such an algorithm is not cryptographically secure [19]. If a sufficient number of PRNs are gathered, an attacker can predict subsequent values.

Lagged Fibonacci generator (LFib). LFib is an improvement of the "standard" LCG, where PRNs are derived as a generalization of the Fibonacci sequence. Such a sequence is generated according to the recurrence relation $S_k = S_{n-j} \otimes S_{n-p} \mod m \ (0 < j < p)$ [14], where $\otimes$ is any binary function such as addition, subtraction, multiplication, or even the bitwise XOR. LFib thus requires an initial sequence and two seeds to begin. However, such algorithms using two seeds can be vulnerable to birthday attacks [3]. Alternatively, therefore, three seeds are recommended to be used according to the expression $S_k = S_{n-q} \otimes S_{n-j} \otimes S_{n-p} \mod m \ (0 < q < j < p)$. In addition, the initial sequence of LFib should contain at least one odd number; otherwise, the generated PRNs would be all even. However, even if these issues are addressed, LFib is still not cryptographically secure because it is represented as a linear recursion.

Mersenne Twister (MT). MT [6] is another popular algorithm. MT does not use any arithmetic operations (i.e., $+$, $\times$, $-$, $\div$) but is based on a group of permutation and tempering operations with shifts ($<<$ and $>>$), AND ($\&$), OR ($||$), and XOR($\oplus$). Since MT is based on a linear recursion, it is not cryptographically secure [2]. One can determine the internal state of the algorithm once a sufficiently long sub-sequence of outputs is observed. The most common implementation of MT is MT19937, in which only 624 distinct outputs can be used to derive all the internal state variables of the PRNG [20].

Well Equidistributed Long-period Linear (WELL). Similar to the MT algorithm, WELL is also a form of linear feedback shift register using simple bitwise operations. A seed value is required to start the generation process. With only a slightly higher time cost, WELL obtains better equidistribution than MT [15]. WELL512, with a state size of 512 bits, is a widely used version. Its generated outputs are only selected within the restricted state instead of unbounded dynamic memory allocation. The period length of the generated PRNs is approximately 2^{512}. At first glance, the use of WELL512 is sufficiently secure for OTP generation. However, when the same seed is used, WELL512 also becomes vulnerable [21].

B. Randomness Functions

Most programming languages provide the functions for generating PRNs by default. Instead of analyzing all of them, we only consider the most common ones — C/C++, Python, PHP, Java, and JavaScript [22]. We introduce the randomness functions that are frequently used to generate PRNs and discuss each function's security issues.

1) C/C++: Many PRNG functions are provided in the C library. Two primary functions are listed below.

rand(): This function is a C standard built-in generator. To ensure that the sequence of PRNs is unpredictable, `srand(·)` is called to initialize the PRNG with a seed value beforehand. The algorithm of `rand()` is adapted from the BASIC PRNG algorithm, and simple operations such as arithmetic (e.g., $\times$) and bitwise operations (e.g., $\&$) are involved. If a static seed is used, the `rand()` generates the same stream of PRNs. Hence, in order to produce an unpredictable sequence of PRNs, the initialization should not be a constant value, but a pseudo-random value instead. Nonetheless, `rand()` uses the LCG algorithm without adding any entropy to the generator, which is not cryptographically secure. Although the LCG parameters

are unknown, the attacker can easily identify the parameters when consecutive outputs are collected.

rand_s(): This is a secure alternative for `rand()`. It generates cryptographically secure PRNs depending on the operating system. It is not affected by the seed produced by `srand()`; it also does not affect the pseudo-random number sequence used by `rand()`. It is essential to mention that this function only works on Windows XP and its later versions.

2) Python: Python uses the MT algorithm as the core generator and leverages the standard MT implementation (i.e., MT19937). Hence, the randomness functions in Python are unsuitable for cryptographic purposes.

random.randrange(start, stop): This function generates a pseudo-random integer within a range of [start, stop]. By default, `start` is defined as zero.

numpy.random.rand($d_0, d_1, \ldots, d_n$): This function takes as input the dimensions of the array to be created. It then creates the array and fills it with PRNs from a uniform distribution over [0, 1).

3) PHP: PHP is the most popular programming language for server-side scripting. Three main functions are discussed.

lcg_value(): This function is used in numerous places within the Zend engine code as an internal function. This function leverages two LCGs to get better quality PRNs. A default seeding algorithm uses time and process id inputs to calculate a value to seed both LCGs. The function implementation is proved to be cryptographically insecure as one can determine the generated values by only using consecutive outputs [2].

rand(min, max): This function generates a pseudo-random integer. By default, it falls back to `rand()` supported by C with the range [min, max] is $[0, 2^{31} - 1]$. The implementation of this PRNG generates a default seed by taking the current timestamp and the value generated by `lcg_value()` as input. Alternatively, users can call `srand(·)` to set the seed externally. When a constant or predictable seed is chosen, this PRNG becomes insecure.

mt_rand(): This function is implemented using the MT algorithm. By implementing MT19937, namely, the standard implementation of MT, this function is not cryptographically secure because the algorithm's internal states can be observed when 624 successive outputs are gathered.

4) Java: There are three primary functions for PRNG implementations in the Java development kit (JDK).

random(): This function is included in the class `java.util.Random` for generating a stream of PRNs with positive signs and restricted within [0.0, 1.0]. This function is created based on Knuth's subtractive algorithm [23], that is, an internal PRNs repeated cycle length is fixed. In addition, this class utilizes LCG with a 48-bit static seed. Without any entropy added to the generator, this function is not cryptographically secure because two pseudo-random sequences created by the same seed are the same. When the PRNG is unknown, the PRNG parameters can be calculated when 2^{32} consecutive outputs are known.

Math.random(): This function is included in the `java.util` package. Without giving any seed, a `java.util.Random` object is created when `Math.random()` is called; hence the PRNs generated by `Math.random()` depend on `random()`. It is also cryptographically insecure.

SecureRandom(): This function is included in the Java class `java.security.SecureRandom` to produce secure PRNs. By using SHA-1 as part of the random algorithm, the generated PRNs are hashed; hence `SecureRandom()` provides a strong PRNG implementation to ensure a non-deterministic output.

In Java, it is thus critical to use the randomness function `SecureRandom` in `java.security.SecureRandom` class instead of using `random()` or `Math.Random()` to generate PRNs for security reasons.

5) JavaScript.: JavaScript also provides `Math.Random()`, which is insecurely implemented in the same manner as Java. However, several cryptographically secure randomness functions are also supported. A web cryptographic function `window.crypto.getRandomValues` with a high entropy seed is recommended. Furthermore, a secure crypto library (SJCL)[1] is also provided.

III. Randomness Rules for One-Time Password

While weaknesses and recommended implementations of the randomness algorithms and functions are precisely described, we are curious whether the PRNGs implemented by developers achieve the expected security level. Therefore, we introduce three generic types of randomness rules for OTP. The sequence of the gathered OTP values that violate any of these rules indicates that the OPT values can be predicted with a significant probability. Note that we assume the network channel is secure when OTP values are transmitted.

1) Rule 1. Do not use a static OTP value [5]: It forbids using a static value for all login sessions of different accounts, which violates the requirement of OTP randomness. Consequently, the security of the OTP authentication scheme is not guaranteed once the OTP value is exposed to attackers.

Threat. When a static OTP value is used, the OTP authentication is vulnerable to replay attacks. If the length of the OTP value is insufficient [12], the OTP is guessable through brute force attacks.

2) Rule 2. Do not generate OTP values according to specific patterns [12] [13]: OTP values should be unpredictable. However, some real-world randomness functions generate OTP values in specific patterns, which can be statistically observed. We discuss the three sub-rules below.

Rule 2-1. Do not generate a repeated sequence of OTP values: It states that the PRNG should not generate PRNs with a fixed period length. Unsurprisingly, if OTP values are periodically repeated, attackers can easily predict the OTP values when the number of generated OTP values is larger than the period size.

Rule 2-2. Do not repeat each distinct OTP value n times: It demonstrates that each OTP value should not be used repeatedly for the n consecutive login sessions.

[1]SJCL: https://crypto.stanford.edu/sjcl/

Rule 2-3. Do not generate OTP values with predictable binary representations: It states that the PRNs should be pseudo-random in any format. Even though the generated OTP values in the decimal format seem unpredictable, they may have some specific patterns in the binary format. For example, the parity of OTP values has a specific pattern such as all evens or $(odd, even, odd, even, \dots)$ parity. In such cases, the possible space of the possible OTP values can be reduced significantly.
Threat. OTP authentication is vulnerable to replay attacks when the generation pattern of OTP values is exploited. The attacker first collects a certain number of the login communication packets. Without retrieving the plaintext of the OTP values, the attacker can send the corresponding packet to the server consistent with the generation pattern.

3) Rule 3. Do not use a constant or predictable seed to initialize a randomness function [24]: It states that the randomness functions should not be seeded with a constant or predictable seed. When the seed is a constant value, the attacker can duplicate the sequence of PRNs when the seed is guessed. For example, when using `srand(1)` to seed `rand()`, `rand()` always outputs the same sequence because the initialization status is determined. Note, when a dynamically changed seed is used for the randomness function, the OTP authentication can still be insecure if the seed is predictable (e.g., use of a timestamp) [25].
Threat. When a randomness function (e.g., `rand()`) is seeded by a constant or a predictable value, the attacker can test the possible seeds through brute force attacks and infer the following OTP values.

IV. Challenges

Three challenges need to be addressed while analyzing the real-world apps to identify the implementations violating the randomness rules introduced in Section III.
Challenge 1: How can we determine the selected randomness algorithm and the PRNG implementation resided on the server-side? App developers typically select a particular randomness algorithm to generate OTP values. However, general program analysis techniques such as program slicing [26] cannot be used here. We cannot access the OTP generation implementation because it resides on the server-side. Diverse PRNG implementation options (written in many different programming languages) make it harder to decide which PRNG is specifically used for the OTP functionality.
Challenge 2: How many OTP values should be gathered to infer potential patterns in an OTP sequence? A large number of OTP values are needed to determine whether any pattern exists in the OTP sequence. Therefore, multiple login attempts should be made. However, it is time-consuming or often not allowed to gather a massive number of OTP values through login attempts. Therefore, we need to minimize the number of login attempts as much as possible.
Challenge 3: How can we collect OTP values for our experiments without affecting the OTP server? We can perform the login process repeatedly to collect a sufficient number of OTP values. However, sending a large number of login requests can interfere with the OTP server's normal operations. More worryingly, it can raise ethical issues.

To address these challenges, we propose three approaches: 1) We analyze the existing PRNG implementations written in diverse programming languages and then abstract the common implementations in Android apps. 2) Based on the analysis of existing PRNG implementations, we obtain vulnerable codes in each code snippet. If a predictable pattern is found, we then determine how many OTP values are sufficient to infer the following PRNs. 3) We set a maximum number of login attempts for collecting OTP values because we need to stop the OTP data collection process when we fail to identify a specific pattern from a sequence of OTP values. Moreover, since some apps limit the number of login attempts per day (e.g., 20 per day), it would be time-consuming to collect the number of OTP values larger than such a maximum number of login attempts. Therefore, we empirically set the maximum login attempts as 1,000 by considering both practicality and efficiency.

V. Overview of OTP-Lint

To investigate the randomness of OTP values, we build `OTP-Lint` to collect a sufficient number of OTP values from a given Android app using the SMS OTP authentication. `OTP-Lint` then checks whether randomness rules listed in Section III are violated. Figure 1 shows the workflow of `OTP-Lint`, which includes three components: *Authentication Locator*, *Request Processor*, and *Vulnerability Detector*.

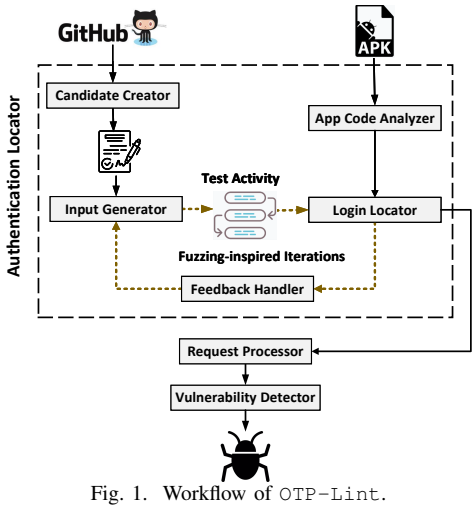

Fig. 1. Workflow of `OTP-Lint`.

A. Authentication Locator

In order to trigger login requests for OTP collection, `OTP-Lint` first identifies whether there are any login *Activities* implemented in apps. The variety of login *Activities*, which are named differently, makes it challenging to identify login *Activities* by simply using keywords matching [27] [5]. The customized functions are generally declared by using abbreviations and informal terms (e.g., `AccountAct`, `AuthAccount`), which are difficult to recognize.

For two functions using similar keywords, these functions' semantics might be different if the order of these keywords

in the two names is different. As an example consider the functions `SMSLoginAct` and `LoginSMS`. They have different semantics although both of them utilize the keywords `SMS` and `login`. The `SMSLoginAct` function represents a SMS login *Activity*, whereas function `LoginSMS` requires the OTP value for identity verification. Therefore, only matching the keywords makes the *Activities* identification inaccurate.

In order to achieve an accurate login *Activity* identification, `OTP-Lint` applies a fuzzing-inspired approach. Fuzzing [28] is an automatic test generation and execution to find security vulnerabilities. We aim to combine fuzzing with program analysis to address the issues of inconsistent function naming and ambiguous function semantics. Instead of matching keywords, `OTP-Lint` relies on code dependencies to understand the purpose of each function. In particular, it creates a test Activity consisting of dependencies to locate login *Activities*. According to the previous test executions' feedback, `OTP-Lint` dynamically optimizes the dependencies in the test Activity for the next round of *Activity* identification.

The authentication locator executes five steps: 1) a **candidate creator** generates candidate samples; 2) an **app code analyzer** analyzes the code of the target app and extracts dependencies; 3) an **input generator** creates a test Activity; 4) a **login locator** identifies login *Activities* from the real-world Android apps; and 5) a **feedback handler** optimizes the test Activity when the login *Activity* is not identified in an app. `OTP-Lint` executes steps 3–5 iteratively until the number of iterations exceeds a threshold, which indicates that there would likely be no login *Activity* implemented in the apps. Referring to the previous fuzzing approaches [29] [30], we set the iteration threshold as 1,000 by considering both effectiveness and efficiency.

1) Candidate Creator: Candidate samples are a group of sample codes used as references when `OTP-Lint` is creating the test *Activity* for further login *Activity* identification. However, it is difficult to determine whether a piece of code is relevant to user authentication. Hence, `OTP-Lint` obtains the authentication code from GitHub repositories.

Specifically, `OTP-Lint` first crawls all the repositories by searching for the keyword `authentication` and selects those whose read.md and the title contain the keyword. As login *Activities* in Android apps are typically written in Java and the invoked functions are Java methods, `OTP-Lint` initially takes as input the GitHub repositories for authentication [31]. As Android apps are generally written in Java, we only consider the repositories in Java. In total, 9,134 GitHub repositories were analyzed to construct a candidate list.

Instead of analyzing the entire project, `OTP-Lint` only extracts the code snippets that are relevant to authentication. Each candidate is represented according to the format of $C = (tx_1, tx_2, \ldots, tx_n)$, where C is the authentication method name and $(tx_1, tx_2, \ldots, tx_n)$ are the invoked functions. Two steps are proceeded to construct each candidate:

Step 1: Method Determination. To identify authentication code snippets, we follow the same natural language processing technique used by NLP-EYE [27]. First, we manually recognize the login-related classes from the GitHub repositories to construct a reference set. Then we use all posts on Stack Overflow[2] (https://stackoverflow.com/) to generate a code corpus. Given each method name in a project, `OTP-Lint` compares it with the words in the reference set and uses the code corpus to measure the semantic similarity between the method name and the word in the reference set. If two words are regarded as similar, then the method is labeled as an authentication method.

Step 2: Dependency Construction. After identifying the authentication method, `OTP-Lint` derives the in-method and external-method correlations by conducting the intra- and inter-dependency analyses. To extract the in-method correlations, `OTP-Lint` constructs an intra-dependency graph to extract in-method correlations by exploring the invoked functions. Nodes in the intra-dependency graph and the inter-dependency graph are the invoked functions, and each directed edge represents a dependency from a caller function to a callee function. By creating the inter-dependency graph, `OTP-Lint` obtains the external-method correlations. It locates where the customized function is declared and constructs an intra-dependency graph for the customized function. Through the customized function, its intra-dependency graph is connected with the graph of the authentication method.

Since some Java functions that execute common behaviours (e.g., `nextLine()`, `toString()`) and exception handling functions (e.g., `printStackTrace`) might affect the detection precision, `OTP-Lint` further removes these redundant functions.

2) App Code Analyzer: `OTP-Lint` takes each Android app as input and proceeds two steps: *Decompilation* and *Dependency Construction*. First, `OTP-Lint` uses JEB Android decompiler [32] to decompile each app into Java source code. Similar to the candidate creator's dependency construction step, `OTP-Lint` constructs the intra- and inter-dependency graphs to extract the in-method and external-method correlations. Since we do not know which methods are relevant to authentication, `OTP-Lint` targets the entire app code to conduct the dependency analysis. Again, the redundant functions are excluded in the generated dependency graphs.

3) Input Generator: `OTP-Lint` creates a *test Activity* that can be adapted to search for the authentication login *Activity* in each app. Instead of manually defining the test Activity, which would be time-consuming and inaccurate, the input generator constructs and optimizes the test Activity by using a fuzzing-inspired approach.

A test Activity consists of the functions that are commonly invoked for authentication login, represented according to the notation $A = (fc_1, fc_2, \ldots, fc_x)$, where A is a login Activity represented by a two-tuple $(Name, ArgumentNames)$ and $(fc_1, fc_2, \ldots, fc_x)$ are a sequence of function names that must be invoked for authentication. `OTP-Lint` optimizes the test Activity used for the previous iteration to generate the

[2]Stack Overflow is the most popular web service to programmers for discussing technical issues, in the form of Question and Answers.

next test Activity. It takes the following two steps. Note that OTP-Lint randomly selects a candidate from the candidate list as initial input for the input generator.

Step 1: Activity Selection. OTP-Lint reconstructs the test Activity A by referring to the candidate samples. By comparing A with the candidate samples, OTP-Lint finds the candidate C_{sim} that is the most similar to A and then selects the functions in C_{sim} to replace the inappropriate functions in A. Such a reconstruction follows two principles:

- Candidate Selection – Since every single word generally covers one functionality to ensure code readability [33], we assume that two methods with similar method names may fulfill the same functionality. Therefore, OTP-Lint pinpoints C_{sim} by extracting the longest common substring (LCS) [34]. Considering the length of the LCS as the similarity score between A and each candidate, OTP-Lint selects C_{sim} with the highest similarity score.
- Probability distribution – Although A and C_{sim} are assumed to fulfill a similar functionality, the invoked functions in A and C_i might be different. Hence, OTP-Lint randomly selects only one function tx_j in C_i to replace the function in A at the same position during each iteration.

Step 2: Argument Creation. After selecting the functions in the test Activity selection, OTP-Lint fills in the required arguments for the test Activity. With the matched candidate C_i, OTP-Lint identifies the arguments that do not exist in A and then inserts the corresponding argument names.

4) Login Locator: Given the test Activity and the dependency graphs of each Android app, OTP-Lint locates the login *Activity* via *Activity Searching*. Since we only consider the *Activities* defined in each app, OTP-Lint selects the dependency graphs that illustrate the *Activities*. Each dependency graph will be compared against the test Activity to verify whether there is any dependency graph that contains a subgraph, which is isomorphic to the test Activity. A graph is considered relevant to the login *Activity* if a subgraph is identified; otherwise, OTP-Lint selects the next dependency graph.

When all the dependency graphs are analyzed, and none of them are defined as login *Activity*, OTP-Lint continues the following steps to optimize the test Activity. Alternatively, OTP-Lint reports the identified login *Activity* for further analysis.

5) Feedback Handler: To optimize the test Activity, OTP-Lint analyzes the similarity between the test Activity and each dependency graph. Since the test Activity is composed of functions named variously by different developers, OTP-Lint compares the test Activity A and the dependency graph $G = (g_1, g_2, \ldots, g_m)$ function by function.

OTP-Lint conducts a transformed pairwise comparison [35] to compare each fc in A with all g in G. It first computes the similarity score of two functions by using the length of LCS and then constructs a similarity set, which consists of the similarity scores of fc. While comparing fc_i and g_j, OTP-Lint only considers fc_i and g_j as similar if and only if the length of LCS is higher than a threshold LCS_{thresh}. Otherwise, OTP-Lint follows the dependency in G to compare fc_i with the rest functions, i.e., $[g_{j+1}, g_m]$.

Among the similarity sets, OTP-Lint chooses the set with the highest total score and replaces the corresponding function with the lowest similarity score in the next iteration.

B. Request Processor

After locating the login *Activity* in each app, OTP-Lint follows the steps used by AUTH-EYE to identify the app with the SMS OTP authentication scheme and then triggers the login request button to retrieve OTP values from the app server.

Step 1: SMS OTP Location. For each SMS OTP authentication scheme, the specific widgets (i.e., EditText and Button) are required. Therefore, OTP-Lint identifies the SMS OTP authentication scheme by recognizing whether there exist relevant widgets. Since widgets used in apps are named typically, which is easier for users to recognize, OTP-Lint performs keyword matching directly to identify the necessary widgets. We created a keyword list containing the keywords such as "sms" and "mobilephone". To extract the widget information (i.e., type, text, orientation, and layout), OTP-Lint utilizes UI Automator to parse the XML layout files. Then it matches the keywords with the text in the field of `android:text` to identify an SMS OTP login. As a result, a list of apps that implement SMS OTP authentication is retrieved.

Step 2: OTP Extraction. In order to send OTP requests automatically, OTP-Lint executes Monkey[3] to fill in the mobile phone number and trigger the button to start a login attempt. However, Monkey only generates pseudo-random streams of user events without locating where the corresponding widgets are. In terms of the collected widget information (e.g., layout, type), OTP-Lint uses UI/Application Exerciser to locate EditText and Button and calls the `dispatchString` method to enter a valid phone number. Because the Android phone involved in the experiment is rooted, OTP-Lint obtains the returned SMS messages from the dataset `/data/data/android.providers.telephony/databases/mmssms.db`. Finally, it parses the SMS messages to extract the OTP values.

C. Vulnerability Detector

OTP-Lint evaluates whether randomness of OTP values produced by real-world Android apps violates the randomness rules summarized in Section III. We set the waiting time interval between two login requests to one minute.

1) Rule 1: Do not use a static OTP value: To identify whether a static OTP value is used, OTP-Lint first requests five OTP values and checks whether they are the same. If so, OTP-Lint then requests 15 OTP values to check whether the value changed[4]. OTP-Lint labels an app generating

[3]UI/Application Exerciser Monkey: It acts as a stress test on the developed app, downloaded from https://developer.android.com/studio/test/monkey

[4]In many apps, the maximum number of login attempts allowed per day is 20.

static OTP values when the 20 OTP values are the same. When an app renews the OTP value within the 20 times of login requests, such a vulnerable situation is discussed in Rule 2-2. In this paper, we only retrieve OTP values without consuming them. However, we observed that some apps would only generate a new OTP value when the previous value is consumed. Such a scheme is also insecure because attackers have time to launch attacks before the OTP value is consumed.

2) Rule 2: Do not generate OTP values according to specific patterns: Without the knowledge about the PRNG implementation on the server-side, we downloaded the sample codes of the PRNG implementations shared on GitHub [17] and Stack Overflow [18] as references. Then we identify a range of implementations that violate the randomness rules.

Rule 2-1: Do not generate a repeated sequence of OTP values. Referring to the PRNG functions introduced in Section II, we select the vulnerable implementations and identify the fixed repeat length of the PRNG. According to the sequence length, `OTP-Lint` sends a sufficient number of login requests to check whether the OTP values are repeated after a certain number of values are generated. In particular, when a PRNG only generates N distinct PRNs (i.e., the fixed repeat length of the PRNG is N), `OTP-Lint` sends $2 \times N$ OTP requests and then compares the sequences $\{1_{st}, 2_{nd}, \ldots, N_{th}\}$ and $\{(N+1)_{th}, (N+2)_{th}, \ldots, 2 \times N_{th}\}$. If two sequences are identical, `OTP-Lint` labels the PRNG as insecure.

Rule 2-2: Do not repeat each distinct OTP value n times. `OTP-Lint` examines generated OTP values to determine whether an OTP value is repeated consecutively (i.e., for n consecutively login sessions, where $n = 2, 3, \ldots$). The PRNG is labeled as insecure when an OTP value is repeated n times.

Rule 2-3: Do not generate OTP values with predictable binary representations. Some OTP values seem to be pseudo-random in their decimal formats. While analyzing the sample code, we found that some developers create their PRNGs using shift operations instead of invoking the corresponding APIs. Thus, `OTP-Lint` converts the decimal format into the binary format to check whether there is any pattern in generated PRNs.

Since various shift operations can be utilized, it is challenging to determine generally about what operations are carried out. We only focus on the simple shift operations (i.e., in clockwise/anticlockwise). First, `OTP-Lint` obtains an OTP value and transforms the value into its binary format. According to the number of digits included in the binary value, `OTP-Lint` requests the corresponding number of OTP values and retrieves their binary values. By comparing the i^{th} and the $(i+1)^{th}$ values, `OTP-Lint` identifies whether the i^{th} value can be transformed to the $(i+1)^{th}$ value through digit shifting (either forward or backward). Additionally, we also observed that the digits in some sequences do not shift iteratively, but a new digit ("1" or "0") is inserted at the end of the sequence. Therefore, `OTP-Lint` only analyzes how the digits of the first binary value shift instead of considering the newly inserted digits. If a shift operation is recognized, the PRNG is labeled as insecure.

For the OTP sequences whose vulnerable patterns are not identified, `OTP-Lint` checks the last digit of each OTP value. If the last digit is '0', `OTP-Lint` labels it as odd; otherwise, it labels it as even. To determine whether a stream of pseudo-random values is parity-guessable, we send the login requests 20 times. If the received 20 OTP values appear by following a specific parity pattern, `OTP-Lint` labels the PRNG as insecure.

3) Rule 3: Do not use a constant or predictable seed to initialize a randomness function: Because the same stream of PRNs can be generated when a static value is used as the seed of PRNGs, we manually inspect the PRNG sample code to learn what static values are frequently used. As a result, the `rand()` functions in C/C++ and PHP are commonly seeded by `srand(1)`; thus we simulate the PRNG by using `srand(1)`. First, `OTP-Lint` sends 1,000 login requests to retrieve 1,000 OTP values by considering both ethnic and efficiency issues (refer to Section IV). It then analyzes the length of the OTP values and executes the PRNG simulation to generate 50 PRNs in the same length. If the sequence of the PRNs is a subsequence of the OTP sequence, `OTP-Lint` labels the app PRNG as insecure.

Apart from that, we found that many posts recommend developers to seed randomness functions by using a timestamp (i.e., srand(time(0))). Therefore, we create a PRNG simulation by using a flexible seed. To ensure that the same timestamp is used for the simulated PRNG and the PRNG in an app, `OTP-Lint` executes the simulated PRNG and simultaneously sends the login requests. Then it determines whether the values generated by the simulated PRNG and the app PRNG are the same.

VI. Evaluation

Our evaluation has two goals. The first is to conduct a large scale randomness analysis on OTP values. The second is to inspect the analysis result manually to understand the detail implementations of these vulnerable PRNGs.

A. Dataset

We downloaded 6,431 top list apps from both GooglePlay and Tencent MyApp market (1,000 from Google Play and 5,431 from Tencent MyApp)[5]. When an app is existed in both app stores, we assumed the implementations of the both apps are the same and removed the one from the Tencent MyApp dataset. The apps are selected from 21 categories[6] and the 300 top listed apps in each category are obtained.

B. OTP Login Activity

`OTP-Lint` successfully analyzed 4,015 out of 6,431 apps. The failed cases are discussed in detail in Section VI-E. Through the fuzzing-inspired approach, `OTP-Lint`

[5]We found that the apps published on GooglePlay barely use OTP authentication; thus we focus more on the Tencent MyApp market.

[6]Categories: Communication, Education, Health & Fitness, Medical, Books & Reference, Photography, Productivity, Video Players & Editors, Travel & Local, Map & Navigation, Entertainment, Lifestyle, Shopping, Tool, News & Magazine, Personalization, Productivity, Social, Beauty, Finance, Music & Audio, and Parenting

found 3,657 apps with login *Activities*. Among these apps, OTP-Lint further identified 2,022 (55.29%) apps with SMS OTP authentication; thus, the following experiments were conducted on these 2,022 apps.

By manually inspecting the apps with SMS OTP authentication, we found that only 214 were from GooglePlay, and all of them utilize two-factor authentication (i.e., using both password-based authentication and OTP authentication). The rest 1,808 apps are from Tencent MyApp; of these apps 1,068 (59.1%) implement two-factor authentication, and 740 (40.9%) only use a single OTP authentication scheme. Since we only analyzed the randomness of OTP values, the parts related to password authentication, i.e., username and password, were filled in manually. OTP-Lint then only executed the OTP login procedure. For the experiments, we registered an account manually in advance for the each tested app.

C. Results

We report the randomness analysis result in Table I. In total, 399 (19.7%) apps out of the 2,022 apps exhibit of violating the randomness rules we have introduced.

TABLE I
VIOLATIONS OF RANDOMNESS RULES.

Violated Rules		# of Apps
Rule 1		41
Rule 2	Rule 2-1	162
	Rule 2-2	67
	Rule 2-3	125
	Total	354
Rule 3		4
Total		399 (out of 2,022)

1) Rule 1: Do not use a static OTP value: OTP-Lint detected 41 (10.3%) apps that produce OTP values violating Rule 1. That is, those apps' PRNGs use a static OTP value instead of generating dynamically modified OTP values for different login sessions. Since only one account was created for each app and the received OTP values were not consumed, we were interested in finding out: 1) whether the OTP value changes after it is consumed; and 2) whether the apps produce the same OTP value for all accounts.

We then conducted two experiments: 1) consuming each OTP value after it is received; 2) registering an additional account and then sending the login requests through the same procedure. The first experiment showed that 20 apps kept returning the same OTP values, although the values were consumed, which indicates that OTP authentication in these apps is mistakenly implemented as password authentication. Due to the short length (typically in four to six digits) and simplicity of the static value, the value can be easily cracked through brute force attacks. For the other apps, which generate new OTP values, we still regard the PRNGs of these apps as vulnerable because a sufficient time window is left for attackers to retrieve the OTP value. We further compared the OTP values generated for the additional account with the original OTP values. All the apps returned a different OTP value for a different account. Therefore, we inferred that the generated OTP values are bound to each user account uniquely.

2) Rule 2: Do not generate OTP values in specific patterns: Among 399 vulnerable apps, 354 apps (88.7%) are categorized as apps using OTP values in a specific pattern. Once the pattern is recognized, an attacker can launch replay attacks to access the victim's account.

Rule 2-1: Do not generate a repeated sequence of OTP values. OTP-Lint detected 162 (40.6%) apps that produce the OTP values violating Rule 2-1. In our current implementation of OTP-Lint, we only considered the PRNGs using MT19937 because it is widely used as a default randomness algorithm in several programming languages such as Python and PHP. Therefore, we set $N = 624$ to test apps because MT19937 and its variants produce PRNs at a fixed period length (i.e., 624) (see Section II-A). Each of their PRNGs generated a pseudo-random stream with 624 unique PRNs and then repeated the same stream. Apparently, the PRNGs of these apps use MT19937 as the randomness algorithm. To verify our result, we further executed a seed recovery tool, untwister[7], to reverse the PRNG and obtain its original seed. With enough PRNs inputs, untwister successfully obtained all their seeds with 100% confidential.

In this paper, we only inferred whether the randomness algorithm of Mersenne Twister (MT) is implemented in the PRNG. OTP-Lint can be extended easily when we exploited the fix period length from the other randomness algorithms and then OTP-Lint can help with the OTP randomness analysis.

Rule 2-2: Do not repeat each distinct OTP value n times OTP-Lint detected 67 (16.8%) apps that produce the OTP values violating Rule 2-2. That is, the detected apps iteratively generated the same OTP value n times. According to our manual inspection, we found that 39 apps repeated each OTP value twice, and 27 apps repeated each value three times. Only one app kept repeating the OTP value for five times.

Since there is no randomness algorithm that can generate a value for n consecutive times and then renew the value. We assume that such a scheme is implemented intentionally and the OTP value are stored and being reused when requested. Therefore, we are curious: 1) how long the OTP value will be stored; 2) whether the OTP value changes after being consumed. Firstly, we conducted three experiments to test how long the OTP value will be stored. Originally, OTP-Lint requested each OTP value after only one minute. We then separately set the waiting period between two login requests as two minutes, 20 minutes and an hour and these OTP values were not consumed. Referring to the previous analysis result, all vulnerable apps repeated the OTP values for less than five times; hence we only sent six login requests in this experiment. Surprisingly, the results showed that 44 apps renewed their OTP values within 20 minutes and six apps generated new OTP values in an hour. For the rest 17 apps, they did not update the OTP values unless the values were consumed.

[7]untwister: https://github.com/altf4/untwister

Afterwards, we ran three similar experiments, in which the waiting periods were set as the same, but we consumed the OTP value after received. The result demonstrated that 62 apps updated their OTP values immediately after the values were consumed (within one minute). For the rest five apps, one of them updates the OTP value within 20 minutes and the other four apps generated new OTP values in an hour.

Rule 2-3: Do not generate OTP values with predictable binary representations. `OTP-Lint` detected 125 (31.3%) apps that produce OTP values violating Rule 2-3. That is, we can predict the OTP values generated from those apps by converting them to binary formats. Among these vulnerable apps, `OTP-Lint` specifically identified 35 apps generating OTP values with certain shift patterns. Through our manual inspection, six apps iteratively shift all binary digits in anticlockwise direction. As an example, an app generated a sequence of OTP values as $< 081642, 032213, 064426, \cdots >$; then, when the numbers are converted from the decimal format into the binary one, the first digit of the current value always appears in the last digit of the next OTP value, i.e., $<\underline{1}0011111011101010$, $0011111011101010\underline{1}$, $0111110111010101\underline{0}$, $\ldots>$. Instead of shifting the binary digits iteratively, 12 apps add either '1' or '0' at the end of the binary sequence. We also found 17 apps that use similar substitution operations on the digits in other positions.

Besides, `OTP-Lint` also exploited 90 vulnerable apps that generate OTP values with "odd-even" patterns. Within these apps, `OTP-Lint` discovered eight apps that only generate even OTP values. We can only infer that their developers might implement the randomness algorithm of LFib, MT, or WELL. However, it is difficult for us to identify the use of a specific algorithm in detail from these OTP sequences without accessing the source code of each PRNG.

3) Rule 3: Do not use a constant predictable seed to initialize a randomness function: `OTP-Lint` detected 4 (0.01%) apps that produce the OTP values violating Rule 3 – the PRNGs of three apps are written in C/C++; the PRNG of the other app is written in PHP. When calling the randomness function `rand()`, three apps use a constant `srand(1)` to seed `rand()`. For the two apps whose PRNGs are written in C/C++, they generate exactly the two same pseudo-random streams. Therefore, attackers can rebuild the PRNG to extract the sequence of OTP values. Surprisingly, we identified one app utilizing the timestamp `srand(time(NULL))` to seed `rand()`.

D. Insights

We manually inspected all the analysis results and gained some insights that we report below.

App Store Comparison. As the apps are downloaded from Google Play and Tencent MyApp, we now discuss the security of the apps collected from each store. Table II presents the number of vulnerable apps collected from each store. Among the 214 apps that were downloaded from Google Play, `OTP-Lint` detected a randomness vulnerability in 137 (64%) apps while it detected a randomness vulnerability in 262 (14.5%) apps out of 1,808 apps downloaded from Tencent MyApp. At first glance, the Google Play apps would be more vulnerable compared with the Tencent MyApp apps. However, all the Google Play apps use a two-factor authentication scheme, which means that the additional user credential information is needed to attack such an app from Google Play. In contrast, 194 (74%) out of the 262 vulnerable Tencent MyApp apps use the OTP authentication alone without any additional security mechanisms, leading to insecure authentication against guessing attacks and replay attacks.

TABLE II
OF VULNERABLE APPS COLLECTED FROM TWO STORES.

Violated Rules		# of Google Play Apps		# of Tencent Myapp Apps	
		1-factor	2-factor	1-factor	2-factor
Rule 1		–	0	11	30
Rule 2	Rule 2-1	–	73	79	10
	Rule 2-2	–	48	10	9
	Rule 2-3	–	15	94	16
	Total	–	136	183	35
Rule 3		–	1	–	3
Total		–	137 (out of 214)	194 (out of 1,808)	68 (out of 1,808)

Furthermore, we specifically examine the difference between both app stores in terms of each randomness rule. For Rule 1, `OTP-Lint` discovered 41 vulnerable apps from Tencent MyApp only. For Rule 2, we can see a different distribution between two stores – among the 136 vulnerable Google Play apps, Rule 2-1 (73 violations), Rule 2-2 (48 violations), and Rule 2-3 (15 violations) are most frequently found in order while among the 218 vulnerable Tencent MyApp apps, Rule 2-3 (110 violations), Rule 2-1 (89 violations), and Rule 2-2 (19 violations) are most frequently found in order. For Rule 3, there are only a few vulnerable apps (1 in Google Play and 3 in Tencent MyApp) commonly in both app stores. The use of the Chi-square homongeneity test [36] revealed that there is a significant difference in the ratios of violated rules between two app stores ($p < 0.05$, $\chi^2 = 169.827$, $df = 4$).

App Category Security. As well as analyzing from which app stores the apps are from, we additionally analyzed the vulnerable apps by categories. We discovered that the top three most vulnerable categories are Video Players & Editors, Entertainment, and Music & Audio. The numbers of vulnerable apps from those categories are 182 (45.61%), 71 (17.80%), and 34 (8.52%), respectively. Although these apps might not be as crucial as apps such as Finance and Social, these apps also store private and sensitive user data. For example, the user often use the Video Players & Editors to edit her private video clips and save them as drafts in the account. The video clips in the draft box are stored in the cloud storage sometimes instead of the user's local storage on her mobile phone. From such Video Players & Editors apps, the attacker can easily access the user's account and steal the stored private video clips.

E. Limitations

Uncertainty about Involved Parameters. Referring to the analyses in Section II, `OTP-Lint` can only determine whether the PRNG of an app is using a known cryptographically insecure algorithm to generate OTP values by analyzing their

patterns. Nevertheless, in some cases, it is unable to decide what PRNG parameters such as the seeds are used for the PRNG implementation on the server-side. To overcome this limitation, we should gather more datasets with various parameter values for PRNGs.

Limited Analysis Scope. `OTP-Lint` detects all violations of randomness rules based on the code samples that are collected from GitHub and Stack Overflow. Since we only focus on specific randomness algorithms and randomness functions based on our codebase, the detection ability of `OTP-Lint` is inherently limited to specific algorithms and functions. However, in practice, there are various PRNG algorithms and implementations that we do not cover in this paper. For example, Clipperz[8] is a Javascript crypto library that we did not consider yet. To evaluate the randomness and predictability of new and unseen PRNGs, we need to introduce more robust statistical tests in the existing PRNG guidelines (e.g., NIST SP 800-22 [37]).

Failed Apps Analysis. `OTP-Lint` failed to decompile or analyze 2,416 Android apps. By manually inspecting these apps, we found that 889 apps use code packing to defend against decompilation. Since their "class" files are encrypted, JEB cannot decrypt the file to extract their source code. In addition, 412 apps have obfuscated code that cannot be analyzed. For 1,115 apps, `OTP-Lint` failed to send login requests because a runtime error occurred while sending request messages.

VII. Related Work

Analysis of PRNG. Several vulnerability analysis studies have been conducted on PRNG. Most of them focus on the PRNG in Linux because it is included in the kernel of all Linux distributions and widely used in many security-related applications and protocols [38] [8] [39] [40]. As the Linux PRNG is open source, Dodis *et al.* [8] and Gutterman *et al.* [38] assessed its security, relying on the code analysis. Dodis *et al.* proposed a new formal security model for PRNGs , which encompassed all the proposed security notions. Then, they evaluated the security of the two Linux PRNGs, `/dev/random` and `/dev/urandom`, and proved that these PRNGs are not robust and do not accumulate entropy properly. Aside from analyzing the source code, Gutterman *et al.* [38] combined static reverse engineering of the source code with dynamic tracing to learn the operation of the PRNG.

Apart from Linux PRNG, OpenSSL is another important application for PRNG analysis [9] [41]. Kim *et al.* [9] investigated the Android OpenSSL PRNG. Similarly, they conducted a code analysis to analyze the Android OpenSSL architecture. Starting from the initialization, they found that every SSL application generates random data from the same initial state. Thus, attackers can recover the state of the OpenSSL PRNG from any apps.

However, the above approaches cannot be applied to analyze the PRNG for OTP authentication because the app PRNG is not open source and typically implemented on the server-side. Therefore, we performed a black-box analysis to detect what algorithm might be implemented and then verified our assumption through dynamic analysis to demonstrate vulnerabilities in the generated stream of OTP values.

Similar to our work, Argyros *et al.* [2] focused on the predictability of password reset token and exploited PHP randomness generators. By analyzing the randomness functions supported by PHP, they launched attacks from the timestamp and the seed aspects, respectively. Without the specific target of the programming language, our analyzing objects are more general.

OTP Authentication. We categorize the previous work on the security of OTP authentication into two groups: vulnerability analysis and security enhancement.

Mulliner *et al.* [1] analyzed SMS OTP authentication from the perspectives of access control and involved parties. They explored the potential weaknesses and introduced attacks to exploit them. Several basic countermeasures were given to defend against the attacks. Instead of analyzing the general OTP authentication, Yoo *et al.* [42] and AUTH-EYE [5] analyzed specific scenarios, i.e., internet banking services, and Android OTP authentication, respectively. Yoo *et al.* investigated the security measures in internet banking and discovered a novel type of attack for hijacking the implemented OTP system. AUTH-EYE presented six OTP rules for the implementation of secure OTP authentication applications. It checked whether OTP authentication violated any of these rules and determined what implementation error existed in the authentication scheme.

Several approaches have been proposed to provide secure OTP authentication methods. Das *et al.* [43] combined the image with numeric values to ensure the unpredictable feature for each OTP. They selected a pseudo-random value as the first part and then randomly selected pixels of user biometric image as the second part. A few studies on OTP authentication schemes focus on specific scenarios. E.g., Jeong *et al.* [44] designed an OTP authentication scheme for home networks with low computation, and Rifa *et al.* [45] designed an OTP authentication scheme for e-banking by combining symmetric cryptography with a hardware security module.

Nonetheless, none of the above approaches assesses the randomness of the generated OTP values. Our work is the first to systemically analyze the randomness of OTP values used in real-world Android apps.

VIII. Conclusion

In this paper, we analyzed randomness algorithms and functions and then introduced three randomness rules that must be followed in the implementation of cryptographically secure pseudo-random number generators (PRNGs). To analyze the PRNG implementations, we designed an automated analysis tool, `OTP-Lint`, to investigate the randomness of OTP values generated by PRNGs. Without accessing the implementation source code, we collected PRNG implementation codes shared by other developers to learn the most common implementations. Finally, we assessed 6,431 real-world Android apps

[8]Clipperz: https://github.com/clipperz/javascript-crypto-library

against the randomness rules and detected 399 apps that generate vulnerable OTP values. Our findings demonstrated that a significant number of existing OTP-based services can be easily attacked in practice, and thus we need to promptly fix those apps to protect users. Perhaps the use of naive PRNGs for OTP authentication is another example of "security theater," which tackles the feeling but not the reality.

ACKNOWLEDGEMENT

This work was supported by the start-up grant of the School of Information Technology and Electronic Engineering, The University of Queensland, and partially supported by the National Natural Science Foundation of China (Grant No.62002222), the National Key Research and Development Program of China (Grant No.2020AAA0107800).

REFERENCES

[1] C. Mulliner, R. Borgaonkar, P. Stewin, and J.-P. Seifert, "Sms-based one-time passwords: attacks and defense," in *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2013, pp. 150–159.

[2] G. Argyros and A. Kiayias, "I forgot your password: Randomness attacks against {PHP} applications," in *Proceedings of the 21st USENIX security Symposium (USENIX)*, 2012, pp. 81–96.

[3] M. Girault, R. Cohen, and M. Campana, "A generalized birthday attack," in *Proceedings of the 8th Workshop on the Theory and Application of of Cryptographic Techniques (EUROCRYPT)*. Springer, 1988, pp. 129–156.

[4] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S & P)*. IEEE, 2017, pp. 154–171.

[5] S. Ma, R. Feng, J. Li, Y. Liu, S. Nepal, E. Bertino, R. H. Deng, Z. Ma, and S. Jha, "An empirical study of sms one-time password authentication in android apps," in *Proceedings of the 35th IEEE Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 339–354.

[6] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *In Journal of ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[7] J. Eichenauer and J. Lehn, "A non-linear congruential pseudo random number generator," *In the International Journal of Statistische Hefte*, vol. 27, no. 1, pp. 315–326, 1986.

[8] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs, "Security analysis of pseudo-random number generators with input: /dev/random is not robust," in *Proceedings of the 20th ACM SIGSAC conference on Computer & communications security (CCS)*, 2013, pp. 647–658.

[9] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of the 20th ACM SIGSAC conference on Computer & communications security (CCS)*, 2013, pp. 659–668.

[10] "Rfc 4086 randomness requirements for security," https://www.ietf.org/rfc/rfc4086.txt.

[11] M. Saito, M. Matsumoto, V. Roca, and E. Baccelli, "Tinymt32 pseudo-random number generator (prng)(rfc 8682)," 2020.

[12] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "Totp: Time-based one-time password algorithm," *In Journal of Internet Request for Comments*, 2011.

[13] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen, "Hotp: An hmac-based one-time password algorithm," *In Journal of the Internet Society, Network Working Group. RFC4226*, 2005.

[14] M. Mascagni and A. Srinivasan, "Parameterizing parallel multiplicative lagged-fibonacci generators," *In Journal of Parallel and Distributed Computing*, vol. 30, no. 7, pp. 899–916, 2004.

[15] F. Panneton, P. L'ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *In Journal of ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 1, pp. 1–16, 2006.

[16] K. H. Brown, "Security requirements for cryptographic modules," *In Journal of Federal Information Processing Standards Publication (FIPS)*, pp. 1–53, 1994.

[17] "One-time password," https://github.com/search?q=one+time+password.

[18] "One-time password," https://stackoverflow.com/search?q=one+time+password.

[19] H. Krawczyk, "How to predict congruential generators," *In Journal of Algorithms*, vol. 13, no. 4, pp. 527–545, 1992.

[20] A. Jagannatam, "Mersenne twister–a pseudo random number generator and its variants," *George Mason University, Department of Electrical and Computer Engineering*, 2008.

[21] "Well512 prng," https://pharoweekly.wordpress.com/2018/11/11/new-random-generator-well512-prng/.

[22] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Proceedings of the 37th IEEE annual computer software and applications conference (ACSAC)*. IEEE, 2013, pp. 303–312.

[23] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, "The emperor's new apis: On the (in) secure usage of new client-side primitives," in *Proceedings of the Web*, vol. 2, 2010.

[24] A. Sinai, "Pseudo random number generators in programming languages," *Master Dissertation*, 2011.

[25] J. Boyar, "Inferring sequences produced by a linear congruential generator missing low-order bits," *In Journal of Cryptology*, vol. 1, no. 3, pp. 177–184, 1989.

[26] M. Weiser, "Program slicing," *In Journal of IEEE Transactions on Software Engineering (TSE)*, no. 4, pp. 352–357, 1984.

[27] J. Wang, S. Ma, Y. Zhang, J. Li, Z. Ma, L. Mai, T. Chen, and D. Gu, "Nlp-eye: Detecting memory corruptions via semantic-aware memory operation function identification," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019, pp. 309–321.

[28] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[29] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 116–126.

[30] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Proceedings of the 13th Network and Distributed System Security Symposium (NDSS)*, vol. 8, 2008, pp. 151–166.

[31] "Authentication," https://github.com/search?q=authentication.

[32] "Jeb decompiler," https://www.pnfsoftware.com/.

[33] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining java class naming conventions," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 93–102.

[34] T. Flouri, E. Giaquinta, K. Kobert, and E. Ukkonen, "Longest common substrings with k mismatches," *The Journal of Information Processing Letters*, vol. 115, no. 6-8, pp. 643–647, 2015.

[35] S. Ma, E. Bertino, S. Nepal, J. Li, D. Ostry, R. H. Deng, and S. Jha, "Finding flaws from password authentication code in android apps," in *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2019, pp. 619–637.

[36] R. A. Alexander, M. J. Scozzaro, and L. J. Borodkin, "Statistical and empirical examination of the chi-square test for homogeneity of correlations in meta-analysis." *In Journal of Psychological Bulletin*, vol. 106, no. 2, p. 329, 1989.

[37] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," Tech. Rep., 2010.

[38] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the linux random number generator," in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2006, pp. 15–pp.

[39] A. Everspaugh, Y. Zhai, R. Jellinek, T. Ristenpart, and M. Swift, "Not-so-random numbers in virtualized linux and the whirlwind rng," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S & P)*. IEEE, 2014, pp. 559–574.

[40] T. Ristenpart and S. Yilek, "When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography." in

Proceedings of the 15th Network and Distributed System Security Symposium (NDSS), 2010.

[41] R. Oak, C. Rahalkar, and D. Gujar, "Poster: Using generative adversarial networks for secure pseudorandom number generation," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 2597–2599.

[42] C. Yoo, B.-T. Kang, and H. K. Kim, "Case study of the vulnerability of otp implemented in internet banking systems of south korea," *In Journal of Multimedia Tools and Applications*, vol. 74, no. 10, pp. 3289–3303, 2015.

[43] R. Das, S. Manna, and S. Dutta, "Secure user authentication system using image-based otp and randomize numeric otp based on user unique biometric image and digit repositioning scheme," in *Proceedings of the 1st International Conference on Communication, Devices, and Computing (ICCDC)*. Springer, 2017, pp. 83–93.

[44] J. Jeong, M. Y. Chung, and H. Choo, "Integrated otp-based user authentication scheme using smart cards in home networks," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2008, pp. 294–294.

[45] H. Rifa-Pous, "A secure mobile-based authentication system for e-banking," in *Proceedings of the 20th Confederated International Conferences On the Move (OTM)*. Springer, 2009, pp. 848–860.