Do Not Give A Dog Bread Every Time He Wags His Tail: Stealing Passwords through Content Queries (CONQUER) Attack

Chongqing Lei[y], Zhen Ling[y] , Yue Zhang[z], Kai Dong[y], Kaizheng Liu[y], Junzhou Luo[y], and Xinwen Fu[x]
[y]Southeast University, Email: f leicq, zhenling, dk, kzliu18, jluog@seu.edu.cn
[z]Jinan University, Email: zyueinfosec@gmail.com
[x]University of Massachusetts Lowell, Email: xinwen_fu@uml.edu

***Abstract*—Android accessibility service was designed to assist individuals with disabilities in using Android devices. However, it has been exploited by attackers to steal user passwords due to design shortcomings. Google has implemented various countermeasures to make it dif cult for these types of attacks to be successful on modern Android devices. In this paper, we present a new type of side channel attack called content queries (CONQUER) that can bypass these defenses. We discovered that Android does not prevent the content of passwords from being queried by the accessibility service, allowing malware with this service enabled to enumerate the combinations of content to brute force the password. While this attack seems simple to execute, there are several challenges that must be addressed in order to successfully launch it against real-world apps. These include the use of lazy query to differentiate targeted password strings, active query to determine the right timing for the attack, and timing- and state-based side channels to infer case-sensitive passwords. Our evaluation results demonstrate that the CONQUER attack is effective at stealing passwords, with an average one-time success rate of 64.91%. This attack also poses a threat to all Android versions from 4.1 to 12, and can be used against tens of thousands of apps. In addition, we analyzed the root cause of the CONQUER attack and discussed several countermeasures to mitigate the potential security risks it poses.**

I. INTRODUCTION

Even with the adoption of biometric authentication methods such as fingerprint scanning and facial recognition, as well as additional hardware authentication methods like USB keys and IC cards, passwords remain an important means of authenticating a user in mobile security. As the first line of defense against unauthorized access, passwords protect valuable user data, making them an attractive target for attackers. While traditional attacks like brute-force attempts are not impossible, they often come with significant drawbacks, like a high time cost. As a result, attackers are seeking more practical methods to obtain user passwords, such as distributing malware (e.g., keyloggers) onto mobile devices, or guessing passwords based on personal information [34].

*Corresponding author: Prof. Zhen Ling of Southeast University, China.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24005
www.ndss-symposium.org

On Android devices, malware often uses the accessibility service to steal user passwords. The accessibility service was originally designed to assist users with disabilities in using Android apps, but it can be easily exploited by malware to collect sensitive information such as passwords. This is because the accessibility service has the ability to interact with victim apps and obtain information such as the content of foreground windows and app life cycles without involving the user. Attackers can use the accessibility service to passively collect user credentials through accessibility events or actively hijack input or output channels to intercept user credentials. Previous research has demonstrated the potential of abusing the accessibility service for malicious purposes, including the collection of user credentials (e.g., [11], [18], [8]) and the interception of user input (e.g., [21], [17], [11], [18]).

Google is aware of the risks posed by accessibility-service-based attacks and has implemented various defenses to mitigate them. One defense strategy is to remove passwords from accessibility events (e.g., [18]). Other defenses aim to increase user awareness by sending warning messages or requiring user confirmation before allowing the accessibility service to access certain information (e.g., [35]). These measures make it difficult to exploit the accessibility service to steal passwords on newer versions of Android.

However, in this paper, we show that existing defenses against password stealing attacks can be bypassed and passwords can still be stolen by exploiting a new query-based side channel — using findAccessibilityNodeInfos-ByText(text) to query the content of the foregrounds. Particularly, this allows users to locate user interface (UI) elements containing specific text by returning a list of such elements. For example, consider an Android app that has a login button with the text “Login”. Using this API, an assistive app can quickly identify the login button and automatically click it for the user. While it is intended to help users navigate and use apps more easily, we have found that it can also be exploited to steal passwords: password fields are also UI elements that can be located by searching for a string that exists within the password. Surprisingly, Android neither prevents searches of password fields (it returns the password input box as usual if the searched string hits the real password) nor alerts users to this potential security risk. To make matters worse, even if only bullet points are displayed in the password field (which is a type of defense enabled by Android), this API can still searches the given text in the real password.

Based on this observation, we introduce the Content Queries (CONQUER) attack, a new query-based side channel method for stealing passwords. CONQUER exploits the side channel by repeatedly querying the latest password character entered using the API as the users types their passwords, by setting the parameter to all possible characters. If the API returns results indicating that the queried string is in the password field, we can determine what the user has just entered and update our collected password accordingly (essentially, we are performing a brute-force attack to query each newly entered password character).

While CONQUER may seem straightforward to launch, there are three challenges that must be addressed in order to make it practical. First, query results can be ambiguous due to the existence of content descriptions (which are configured by the developers) and do not necessarily indicate the presence of specific strings or characters in a password. Second, some apps have implemented defenses to mitigate previous attacks (e.g., [18], [8]). As a side effect, these defenses may hinder an attacker's ability to determine when to perform queries. For instance, these defenses may block outgoing password-related accessibility events (through which the malware can determine when to launch the queries), and malware can no longer receive them. Third, the API being exploited does not require the specified text to be case-sensitive. As such, the malware can only collect a case-insensitive password by default.

Fortunately, we have addressed these challenges and successfully implemented CONQUER. To address the first challenge, we propose a lazy query algorithm: we can combine and query the characters not in the content description but in the password (i.e., lazy queries) to eliminate side effects caused by the text in the content description. To deal with the second challenge, we actively query the password length, and whenever it changes, we know that the user has entered a new character. For the third challenge, we use the time and typing speed as side channels (i.e., pressing the case-switch button will cause more time to enter a character) and set up a state machine that tracks the input state of the user (e.g., pressing the case-switch button) to recover case-sensitive passwords.

We evaluated the effectiveness of CONQUER by conducting extensive experiments. The experimental results on 108 real-world passwords show that CONQUER can recover the original passwords based on the collected case-insensitive passwords and input timing information with an average one-time success rate of 64.91%. We show that CONQUER works against all popular Android versions (ranging from Android 4.1 to Android 12), and all apps that use system-provided password input boxes are therefore vulnerable to our attack. Through our large-scale analysis on 13,786 Android apps that use custom password input boxes, we also identified 13,001 (94.30%) apps that are subject to our attack. We responsibly disclosed our findings to Google, who acknowledged our findings but decided not to fix the issue at this time because the accessibility service requires this behavior to function as intended. We therefore believe that there is no easy fix for CONQUER.

Our major contributions are summarized as follows.

- New Findings. We are the first to propose CONQUER, a novel password-stealing attack that exploits the content query side channel in the Android accessibility service. Though Android currently has multiple countermeasures in place to defend against accessibility-service-based attacks, CONQUER can go around them and steal user passwords.
- New Techniques. To ensure the practicality of CONQUER, we have introduced several new techniques: the lazy query technique eliminates the side effects caused by content descriptions, the length-based side channel allows us to determine if the user is entering the password, and the temporal side channel and state machine enable us to handle case-sensitive passwords.
- Extensive Experiments. We evaluate the performance and security implications of CONQUER through real-world experiments. Our results show that CONQUER can recover case-sensitive passwords with an average one-time success rate of 64.91%. In addition, CONQUER affects system-provided password input boxes on all popular Android versions, and our large-scale analysis of Android apps revealed that 13,001 out of 13,786 (94.3%) apps using customized password input boxes are also vulnerable to our attack.

II. BACKGROUND

In this section, we first introduce the Android accessibility service in §II-A. Next, we discuss existing accessibility-service-based attacks as well as the defenses in §II-B.

A. Android Accessibility Service

The Android accessibility service provides user interface enhancements in assisting users with disabilities (e.g., visual or hearing impairment) [12]. The accessibility service allows apps to be notified of accessibility events triggered by UI actions, such as clicking buttons or scrolling the screen. For example, TalkBack [14] can provide real-time spoken feedback to users while they are interacting with their devices. In order to use the accessibility service, app developers must configure the service to tell the system when and how accessibility service should be invoked, and which event types (e.g., clicking a button or scrolling the screen) the service should respond to. To achieve this, developers need to extend the `AccessibilityService` class and override the callback method `onAccessibilityEvent(event)` to handle received accessibility events and take corresponding actions (i.e., specifying how the app deals with accessibility events).

Being an important service that is open to all developers, the Android accessibility service has offered various APIs. One such API is `findAccessibilityNodeInfosByText(text)`. The API takes a specific text as input and returns a list of `AccessibilityNodeInfo` objects, each representing a foreground UI component containing the specified text. Therefore, this API is typically used to automatically locate UI components that have specific names or contain specific texts. For example, an assistive Android app might use this API to facilitate automatic login. To this end, the app can first get the root node of the currently active window in its target via `getRootInActiveWindow()` (Android organizes UI components in a tree structure, with each active window having a root node [13]). The app can then invoke `findAccessibilityNodeInfosByText("login")`

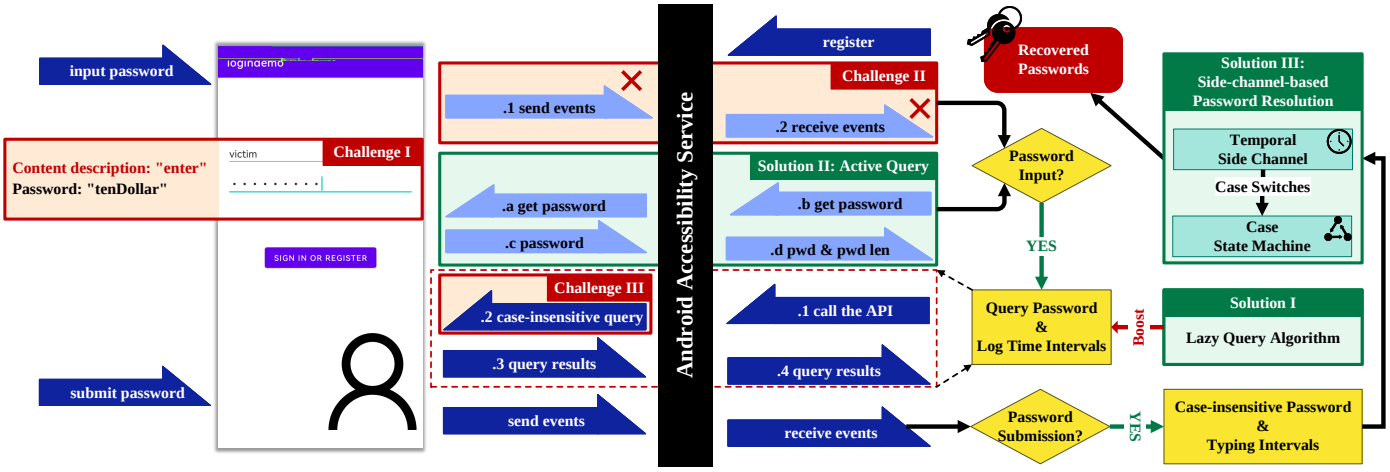


Fig. 1: Overall design of the CONQUER attack

"a", "b", "1"). Consequently, when the attacker feeds the character "p" into findAccessibilityNodeInfosBy-Text(text) , the function returns a list of UI components including the input box. The attacker then knows "p" is the first character of the user's password. Similarly, when the user enters the second character of the password, the password text becomes "pa". If the attacker feeds this string (i.e., "pa") into findAccessibilityNodeInfosByText(text) , the input box will be returned. Please note that before this round of enumeration, the attacker had already determined that the first character of the password is "p", and he or she only needs to attach a single character to the first character to assemble the password string. By using the same method repeatedly, the attacker is able to recover each remaining character of the password.

B. Threat Model and Assumptions

The objective of our attack is to steal passwords from regular users. We exclude individuals who rely on the accessibility service (e.g., visually-impaired individuals), because they often use the touch exploration mode. In this mode, passwords are directly leaked through accessibility events and our attack is not necessary. We make four assumptions for our attack. First, we assume that victims have installed the malware on their Android mobile devices and granted it the accessibility service permission. This assumption is reasonable because many prior studies that target or abuse the accessibility service (e.g., [11]) had similar assumptions. Besides, recent research [7] has revealed that users are willing to grant whatever permissions the apps request if they wish to use them. Therefore, we believe that malware can lure users into granting such permissions unsuspectingly. Second, we assume that victims are using the latest Android phones, as older versions of Android are easier to compromise using attacks introduced in previous efforts [17], [11], [18]. Third, we assume that victims have turned off the "Make Passwords Visible " option in order to prevent characters of the password from being directly extracted through accessibility events. Finally, although the techniques we introduced can be used to deploy other attacks (e.g., obtaining users' keystrokes beyond just their passwords), we particularly focus on password stealing attacks given their high security implications.

IV. OVERVIEW OF CONTENT QUERIES (CONQUER) ATTACK

Based on the observations and assumptions outlined in §III-A, we propose Content Queries (CONQUER) attack, which can be leveraged by the malware to steal user passwords. In the following, we first present the basic workflow of the CONQUER attack, then introduce challenges and the solutions, followed by the overall design of the CONQUER attack as shown in Figure 1.

A. Basic Workflow

At a high level, the basic workflow of CONQUER attack can be broken into four steps: (i) The malware registers (¬) for the accessibility service to receive the intended events (e.g., events triggered by inputting the password). We have omitted the details of these techniques for brevity as they are well-known. (ii) When the user enters the password (-), the malware will get notified (®.1-2) and be able to determine relevant events by inspecting the types of the incoming events (¯). (iii) When the malware determines that the user has entered a specific character of the password, it then extravagantly enumerates possible combinations to infer that character (°) using the observation introduced in §III-A. (iv) When the user hits the login button (±), the malware gets notified (² - ³) and knows that the user has finished entering the password. At this point, the malware has learned the entered password, which forms the output of the malware.

B. Challenges and Solutions

While the attack is theoretically easy to deploy, it still faces various challenges in practice. Generally speaking, there are at least three challenges that attackers must overcome to successfully launch the CONQUER attack. Below, we explain each challenge in greater detail:

(C-I) Differentiating Passwords and Descriptions. The accessibility service has provided content labels to assist users in understanding the meaning of UI elements, and the android:contentDescription attribute is one of these labels. For example, app developers can set the android:contentDescription attribute to "Password"

for a password input box to help users understand that they need to enter their passwords into this field. However, when issuing queries using findAccessibilityNodeInfosByText(text), for UI components such as password input boxes, the content of the android:contentDescription attribute will also be searched along with the actual text (e.g., the entered password). As a consequence, if the content description is set for the password, we cannot determine whether the searched string is in the content description or the password text. For example, assume the password to be entered is "passport", then the strings "p", "pa", and "pas" will exist in both the password (i.e., "passport") and the content description (i.e., "password"). As a result, the password node will be returned in all queries made on these searched texts. Being the attacker, we cannot know whether these strings are part of the password or just part of the content description.

To address this challenge, one intuitive solution is to simply keep all strings indicated to be contained in the password box by the query results, and use these strings as the basis for subsequent queries. For example, queries for each letter in "password" will succeed during the first round of queries. Therefore, we save all these letters as password candidates of length 1 (e.g., "p", "a", "s"). During the second round of queries, multiple enumerations are carried out based on the saved 1-character password candidates respectively, and we further get multiple new password candidates of length 2 (e.g., "pa", "as", "ss"). However, this solution requires more rounds of character enumeration since multiple strings are saved as candidates (instead of only one string under normal cases), which can be extremely time-consuming and the queries cannot finish in time (e.g., before the next character is entered). Our preliminary experiments using this naive method have confirmed this drawback: it can take several seconds to finish one round of queries after a single character input.

Differentiating Passwords via Lazy Queries (§V-A)

This challenge only limits our knowledge of whether characters present in the content description are also present in the password. However, it does not prevent us from querying characters that are only present in the actual password. When we query a character that is not in the content description and it returns the password node, the influence of the content description is eliminated and we can then begin querying the rest of the password (using lazy queries).

(C-II) Breaking Defenses enabled on Victim Apps.Some real-world Android apps do not actively send accessibility events when users enter passwords (i.e., ®.1 in Figure 1 is blocked), even though the app will still respond to inquiries from other apps. This can be achieved by extending the AccessibilityDelegate class and overriding the SendAccessibilityEvent() method. When victim apps block outgoing password-related accessibility events, no app using the accessibility service can receive password-related accessibility events passively. We speculate that this design is intended to protect against previous accessibility-based password-stealing attacks (e.g., [18], [8]). As a result, when targeting these apps, we cannot receive notifications when a victim is entering the password and the associated password node cannot be directly acquired from accessibility events. For example, when users try to log in to Alipay, no password-related accessibility events are sent while they are entering their passwords.

Thwarting Defenses via Active Queries (§V-B)

This challenge only prevents us from knowing when and which object to query passively. However, being an attacker, we can first try to locate the password input box to be queried, then actively query the password length to see if the victim has already started entering the password.

(C-III) Recovering Password from Case-Insensitive Strings. Initially, findAccessibilityNodeInfosByText(text) is designed to assist users with disabilities and has the requirements of being robust. As such, the match is case insensitive containment (e.g., findAccessibilityNodeInfosByText(text) identifies buttons that contain "login" regardless of whether the text "login" is in upper or lower case). This is reasonable: if someone is using TalkBack to receive real-time spoken feedback, he or she will likely not care the case of the letters. However, being a piece of malware that attempts to steal user passwords, it must have the capability of knowing whether the password is in upper or lower case. Intuitively, the malware can enumerate all the combinations of upper and lower case letters in a case-insensitive string (e.g., "Password", "PAssword", etc.). However, it is very costly. For a password that has 8 letters, there could be 64 (i.e., 2^8) combinations. While the attacker can still enumerate those combinations, it is not particularly practical (e.g., many apps will block accounts after several failed login attempts).

Inferring Passwords via Side Channels (§V-C)

We can still try to recover user actions (e.g., switching between cases) by exploiting other side channels. For example, since it usually takes longer to enter the next letter if a user switches case, this temporal side channel can be exploited to detect case switches.

V. Detail Design of Conquer

A. Lazy Query for Password Differentiation

The Lazy Query Algorithm. Since content descriptions are usually short phrases, it is rare for them to cover all characters in the password. Based on this intuition, C-I can be handled by *lazy query*: the content of the password is not queried until users have entered a character that is not in the content description into the password box. As an advantage of this strategy, we can completely eliminate the side effects brought by the content description. The complete lazy query algorithm is shown in Algorithm 1. Let S_c be the set of all (case-insensitive) characters in the content description, and S be the set of all characters that can be used in a password. Initially, we have S_c S. During the lazy query process, every time a character is entered, we first check if the character is not present in the content description by applying single-character

queries on $\overline{S_c}$ (line 7-13). If we fail to get a match, the value of the queried character is ignored. However, the number of ignored characters are recorded (line 25-26). Once we get a match, the password and the content description can be differentiated starting from this character. We then perform backward queries to determine the value of all previously ignored characters by repeatedly performing queries on S_c (line 14-24), and the collected password is finally returned. With the result from the differentiation process, subsequent queries can be handled easily using the normal query method.

Algorithm 1: The Lazy Query Algorithm

```
   Data: set of characters presented in content
         description S_c, set of all possible characters in
         passwords S, the password node N_p
   Result: case-insensitive password P
1  Function main( N_p, S_c, S) :
2      cnt  0;
3      P  '';
4      belazy  True ;
5      while belazy do
6          if a character is entered then
7              for ch 2 (S  S_c) do
8                  if Query(N_p; ch) then
9                      belazy  False ;
10                     P  P + ch;
11                     break
12                 end
13             end
14             if : belazy then
15                 while cnt > 0 do
16                     for ch 2 S_c do
17                         if Query(N_p; ch + P) then
18                             P  ch + P;
19                             cnt  cnt  1;
20                             break
21                         end
22                     end
23                 end
24                 break
25             else
26                 cnt  cnt + 1;
27             end
28         end
29     end
30     return P;
31
32 Function Query( Node, Text) :
33     flag  False ;
34     qr  Node.findAccessibilityNode-
         InfosByText( Text) ;
35     if Node 2 qr then
36         flag  True ;
37     end
38     return flag;
```

For example, consider a scenario where the password we want to query is "tendollar" and the target app is com.infonow.bofa , whose password field has the content description "enter". We first obtain the set S_c = f 'e', 'n', 'r', 't'g from the content description. After the user enters "t", we perform single-character queries on $\overline{S_c}$ and no match is found. The same process is repeated after the user enters "e" and "n", and no match is found as well. However, after the user enters "d", we get a match through single-character queries. We then perform backward queries on S_c to determine previously entered characters: "ed", "nd", "rd" and "td" are queried and we can get a match on "nd". The same steps are repeated until we obtain "tend".

It is worth noting that, although it is very rare, the lazy query algorithm may fail if: 1) the password happens to be a sub-string of the content description; or 2) the content description includes a significant portion of the characters in a password. We cannot handle the first scenario. However, the second scenario can still be handled. The basic idea is to record sub-strings that are only present in the content description, and combine them with characters within the content description to derive new sub-strings that are only possible to be present in the password and query them using the API. Once such a sub-string is found, the password can be quickly identified.

Specifically, we start by defining S_{cj} as the set of all sub-strings in the content description with length j, specifically, we define $S_{c\mathbf{0}} = \varnothing$. By definition, it is clear that $S_{c\mathbf{1}} = S_c$. We further define $S_{nj} = S_{c(j-\mathbf{1})} \quad S_c \quad S_{cj}$, where the Cartesian product symbol represents string concatenation. Whenever a character is entered, single-character queries will first be performed on $\overline{S_c}$ in case a character not in the content description is entered. If it is not the case, assume the current length of the password is k, strings in S_{nk} are enumerated and queried. Since $S_{nk} \backslash S_{ck} = \varnothing$ by definition, if we get a match, the queried string must be the current password and we can fall back to the normal query strategy since then. Otherwise, we know the current password lies within S_{ck}, but the specific value needs to be determined by subsequent queries. If we cannot decide the actual value of the password until password submission, we can only make an assertion that the password is in S_{cl}, where l is the final length of the password. However, since extra queries are needed compared to lazy query, and the size of S_{nj} is large (think about a content description that covers all characters), this algorithm comes with a higher time cost. For example, assume the content description is "enter", but the password is "teren", we first obtain $S_{c\mathbf{1}}$ = f 'e', 'n', 'r', 't'g, $S_{c\mathbf{2}}$ = f "en", "nt", "te", "er"g, $S_{n\mathbf{1}} = \overline{S_c}$, and $S_{n\mathbf{2}}$ = f "ee", "et", ..., "tt"g, etc. When "t" is entered, no match will be found on $\overline{S_c}$ and $S_{n\mathbf{1}}$. The same procedure repeats for the second and the third character. When the fourth character "e" is entered, the current password is "tere" and a match will be found on $S_{n\mathbf{4}}$, and we hence know "tere" only exists in the password.

B. Active Queries for Breaking Enabled Defenses

Even though we will not be passively notified by these

password via N_p.getText().length() (i.e., ®.a-d in Figure 1). As such, we can successfully gather information about when the user has entered a character into his or her password and perform queries accordingly.

We now demonstrate this process using the aforementioned Alipay example (whose package name is com.eg.android.AlipayGphone). First, we locate the password input box and obtain its view ID, which is com.ali.user.mobile.security.ui:id/content . After that, the corresponding AccessibilityNodeInfo object of the password input box can be obtained by invoking findAccessibilityNodeInfosByViewId(id) on the root node of the active window (obtained via getRootInActiveWindow() as discussed earlier) with the parameter set to the acquired view ID. With the collected password node N_p, we can repeatedly request updates on the current length of the password via N_p.getText().length() and determine when we need to perform queries based on the increase of the password length.

C. Side Channels for Passwords Resolution

Intuitively, it may take the users longer to enter the next letter if they switch between upper and lower case letters. This temporal difference can be used as a side channel to detect when the user is switching between cases. However, relying solely on this temporal side channel is not reliable under real-world scenarios for two reasons. First, most keyboards provide two ways to switch cases: using caps-lock or shift, but such information cannot be effectively derived through the temporal side channel. Second, the user may need to switch between different keyboard layouts to enter a more complex password, which, like case switching, also takes longer and hence can be indistinguishable for the temporal side channel. To address the limitations of the standalone time-based side channel and further improve the robustness of the password resolution process, we additionally exploit a state-machine-based side channel to recover characters with better precision (e.g., most keyboards share the same operation logic for case switching, therefore a general state machine can be built to mimic the case switching process). The details of the two side channels are discussed below.

(I) Time-based Side Channel for Case Switch Detection. In order to use the time-based side channel to detect case switches, two challenges must be overcome: First, typing speed varies significantly among people [26], making it difficult to create a universal model that works for everyone. Second, the complexity of passwords can also vary widely, with some requiring users to switch between different keyboards in addition to changing cases. This makes it harder to distinguish between case switches and keyboard switches, as the time intervals between characters may be similar in both cases. For example, consider the password “dot#COm”. To enter this password, the user must switch to the symbols keyboard to enter “#”, then switch back to the letters keyboard and change the case to enter “c”. However, it is difficult to determine whether the user changed the case before entering “c”, as the time interval between “#” and “c” may be large due to the keyboard switch and indistinguishable from a large interval caused by a case switch. As shown in Figure 2, we tackle the two challenges through a two-step process:

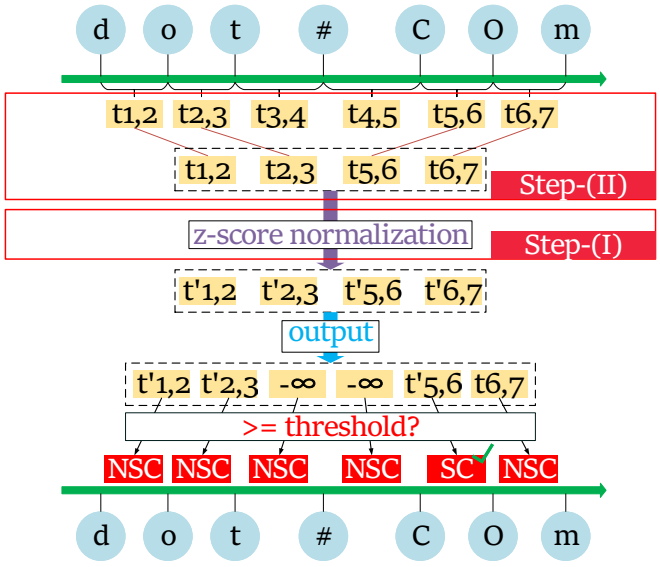


Fig. 2: Workflow of the time-side-channel-based case switch detection

Step-(I): Detecting Case Switches Using Normalized Typing Intervals. To address the first challenge, instead of trying to determine an absolute-time-based model, we build a relative-time-based model using normalized time interval sequences. Normalization can help to reduce individual differences and increase the generality of our model, making it more effective at detecting case switches. Assume the input time interval sequence t for a password with length n is given as $t = (t_{1;2}; t_{2;3}; \dots; t_{n-1;n})$, where $t_{i;i+1}$ represents the time interval between entering the ith and the $(i+1)$th character. Then t is first normalized with z-score:

$$t_{norm} = \frac{t \quad \mu_t}{\sigma_t} \tag{1}$$

where μ_t is the average of t, and σ_t is the standard deviation of t. We further define the true positive rate for detecting case switches and non-case-switches as $TPRCS$ and $TPRNCS$, respectively. With the normalized t_{norm}, a universal optimal threshold value that maximizes $TPRCS$ $TPRNCS$ can be obtained to detect potential case switches.

Step-(II): Improving Robustness by Ignoring Keyboard Switches. To handle the second challenge, our key idea is to only normalize the time intervals between letters. In contrast, all other intervals are discarded before the normalization, and reassigned to the value 1 afterwards. The rationale behind this solution is that by only considering time intervals between letters, we can eliminate the influence of keyboard switching, which can help to create a more robust model. Take the previous password “dot#COm” as an example, the time intervals between “t” and “#”, and “#” and “c” will be discarded during normalization and reassigned the value 1 . Formally, assume $D = \{t_{i_1;i_1+1}; t_{i_2;i_2+1}; \dots; t_{i_k;i_k+1}\}$ is the set of all discarded time intervals, then the original time interval sequence $t = (t_{1;2}; \dots; t_{i_1;i_1+1}; \dots; t_{i_k;i_k+1}; \dots; t_{n-1;n})$

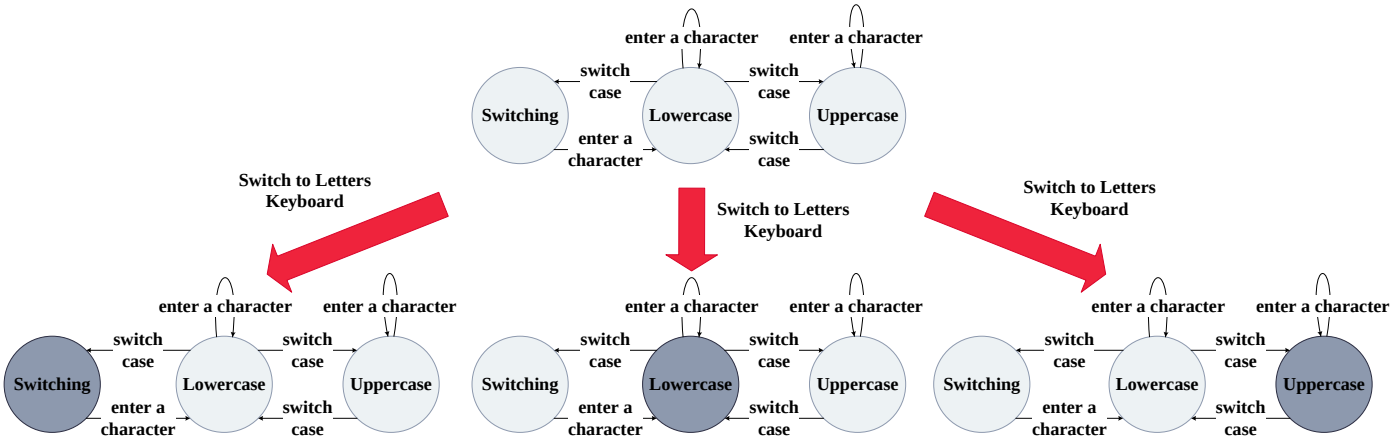


Fig. 3: Case state NFA and state forking

$= (t_{1;2}; \cdots; t_{n-1;n})$ intervals in D Equation (1) on t' to obtain the normalized sequence t'_{norm}

t' is first transformed to

unknown. Therefore, we will have three copies of the initial password (i.e., an empty string) with the case state set to lowercase, switching, and uppercase, respectively.

As the number of case switches in the password grows, the maximum number of possible passwords recovered through the NFA can be shown to form a Fibonacci sequence. To prove this, we first define n_s as the number of detected case switches, and a_n^l as the number of passwords with the case state set to lowercase after n case switches. Similarly, we define a_n^s and a_n^u respectively for the switching and uppercase state. The initial condition is defined by Equation (2). According to Figure 3, the uppercase state will transition to the lowercase state on a case switch, the lowercase state will transition to either the switching or uppercase state on a case switch, and the switching state is temporary, which will transition to the lowercase state once the next character is entered. Therefore, we have the relationship between a_n^l, a_n^s and a_n^u as shown in Equation (3).

$$a_0^l = 1; a_0^s = 1; a_0^u = 1 \tag{2}$$

$$\begin{aligned} a_{n+1}^l &= a_n^u \\ a_{n+1}^s &= a_n^l + a_n^s \\ a_{n+1}^u &= a_n^l + a_n^s \end{aligned} \tag{3}$$

Let Q_n be the number of passwords after n case switches. Through a series of derivations as shown in Equation (4), we can see that $\{Q_n\}$ is a Fibonacci sequence with $Q_0 = 3$ and $Q_1 = 5$.

$$\begin{aligned} Q_{n+2} &= a_{n+2}^l + a_{n+2}^s + a_{n+2}^u \\ &= a_{n+1}^u + a_{n+1}^l + a_{n+1}^s + a_{n+1}^l + a_{n+1}^s \\ &= Q_{n+1} + a_n^u + a_n^l + a_n^s \\ &= Q_{n+1} + Q_n \end{aligned} \tag{4}$$

While the Fibonacci numbers will increase exponentially, it is important to note that Q_n can be guaranteed to be small in most cases. In fact, previous research [20], [23] has shown that even when password composition policies require the inclusion of at least one uppercase letter, the average number of uppercase letters used in passwords is typically no more than 2. With this assumption, even in the worst-case scenario, in which a password contains two non-adjacent uppercase letters that are both located in the middle of the password (therefore 2 case switches are needed), the number of possible combinations is limited to 8 at most given that $Q_2 = 8$.

Step-(II): Dealing with Extraordinary Cases Using State Forking Handler. While the case state NFA is effective at handling continuous letter input, complex passwords may require the user to switch keyboards from and back to the letters keyboard. We handle this situation by proposing a state forking mechanism. The state forking mechanism is activated whenever the user switches back to the letters keyboard (referred to as keyboard switches for brevity), which can be easily detected from the already collected case-insensitive password. During state forking, three copies of the current password with three different case states are created, corresponding to the three different possibilities respectively: lowercase, switching, and uppercase.

To analyze the theoretical password recovery performance with state forking enabled, we further assume that the number of detected keyboard switches is m_s. The number of passwords with the lowercase state after m keyboard switches and n case switches is defined to be $a_{n;m}^l$. Similarly, $a_{n;m}^s$ and $a_{n;m}^u$ are defined for the switching and uppercase state respectively. Now let $Q_{n;m}$ be the number of passwords after m keyboard switches and n case switches. According to Equation (4), we immediately have Equation (5). For every keyboard switch detected, 3 new copies of the password with 3 different case states are generated, therefore Equation (6) holds.

$$Q_{n+2;m} = Q_{n+1;m} + Q_{n;m} \tag{5}$$

$$Q_{n;m+1} = 3Q_{n;m} \tag{6}$$

As shown in our theoretical analysis, the number of guessed passwords heavily depends on the number of case switches and keyboard switches. If the number of case switches and keyboard switches is large, the total number of recovered possible passwords may be up to dozens or hundreds. However, according to our statistical analysis (see §VI-A) on the Rockyou password dataset [28], which consists of 14,344,356 unique passwords and their use counts leaked from a real-world website, shows that for passwords used by 99.65% of the users, the number of case switches and keyboard switches is small. Therefore, the theoretical password recovery performance of our model is sufficient in most scenarios. Our analysis also shows that among users who have passwords with at least one letter, 99.55% of these passwords follow one of three simple patterns: all letters are lowercase, all letters are uppercase, or the first character is the only uppercase letter. Hence, we include these three patterns in the recovered possible passwords to increase the robustness of our attack.

TABLE II: Vendors, device names and OSs of mobile phones used by participants

Vendor	Device Name	OS
Xiaomi	Mi 11	Android 12
	Mi 10 Pro	Android 12
	Mi 10	Android 12
	Redmi K30 Pro Zoom Edition	Android 11
Vivo	IQOO Neo5	Android 12
	IQOO Z1	Android 11
OPPO	Reno5	Android 12
Meizu	16T	Android 9
OnePlus	5T	Android 9
Samsung	Galaxy S8	Android 8.0
Huawei	Mate 20	HarmonyOS 2.0.0
	Mate 30	HarmonyOS 2.0.0
	Mate 40 Pro	HarmonyOS 2.0.0
	P40 Pro	HarmonyOS 2.0.0
	Nova 4	HarmonyOS 2.0.0
	Nova 5	HarmonyOS 2.0.0
Hornor	30 Pro	HarmonyOS 2.0.0
	10 Lite	HarmonyOS 2.0.0

* Note: duplicated (vendor, device name, OS) tuples are ignored.

VI. Evaluation

In this section, we conduct an experimental evaluation of CONQUER. We first illustrate the experiment setup, then

TABLE III: Selected Passwords. “x” represents the number of case switches, and “y” represents the number of keyboard switches.

Category (x, y)	Passwords					
$(0; 0)$	chocolate	password1	butterfly	liverpool	basketball	elizabeth
	tinkerbell	spongebob	alexandra	beautiful	alexander	christian
$(0; 1)$	love4ever	1password	123456789a	1princess	hotmail.com	yahoo.com
	friends4ever	1truelove	1babygirl	love4life	c.ronaldo	4everlove
$(0; 2)$	i love you	2cute4you	2bornot2b	ch0c0late	@hotmail.com	2hot2handle
	2gether4ever	1life2live	l1verp00l	2pac4life	i love me	TEXT ONLY AD
$(1; 0)$	HarryPotter	JesusChrist	ChrisBrown	HelloKitty	LinkinPark	iloveJesus
	TokioHotel	BettyBoop	CyoiydgTv	JohnnyDepp	SpongeBob	HannahMontana
$(1; 1)$	iydgTvmujl6f	iydgTvgl,v	glk;]ydKIN	123qweASD	ry=ibomiN	Tbfkiy9oN
	okiuiy9oN	Lbibiy9oN	vkiuiy9oN	iydgTv,kd	0ydidAKIN	Ibibiy9oN
$(1; 2)$	l6fkiy9oN	iydgTvm6d;yo	iy9ok4iIN	The RockYou Team	0yomiN0bik	4ymik4iIN
	06Rkiy9oN	8ow,jrbgLK	06iuiy9oN	v6[]iy9oN	l64kiy9oN	JOhNY_exstasy_nemis
$(2; 0)$	FallOutBoy	diiIbdkiN	MyChemicalRomance	AaBbCc123	WinnieThePooh	TaeKwonDo
	JesseMcCartney	IchLiebeDich	CrashIntoMe99	dHgTvojkiyd	TeQuieroMucho	SuzieAndRocco
$(2; 1)$	iydotgfHdF’j	dyPPkiy9oN	iydotgfHdF\\’j	l6iLydfbN	v4blbmTbN	obLkiy9oN
	mbrpN;iiI	db99bLydfbN	vii5lbmTbN	oyomNiyd,kiN8	m5vuIbsTd8	gfHd,uxyPsk
$(2; 2)$	ob9bLkl9iN	Mje4nGq6vL45	JaY14$P.rBoricua	2KaEle4cxK	*mZ?9%^jS	y712xC61vIHc
	xXx-rebecca-xXx	xAIvp^jLb]x	pacS*ptt-*KnKA*	l6mTkiy9oN	l6mTb]ydKIN	k6kgWW7WuM

explain the experiment results by answering several research questions.

A. Experiment Setup

Volunteers and Testing Environment. We recruited 20 volunteers to participate in the experiment, which is a typical setup used in previous research (e.g., [11]). The volunteers were recruited from our college campus and were all students. The vendors, device names and OSs of the mobile phones used by the participants are listed in Table II. The experiment was approved by the IRB. The volunteers were instructed to install the provided malware and victim app on their own mobile phones, and to give the malware access to the accessibility service. They were then asked to input several pre-selected real-world passwords into the victim app using their usual keyboards. The case-insensitive passwords and the input time interval sequences collected by the malware during the experiment are later processed to evaluate the efficiency of CONQUER in recovering passwords.

Password Selection. The real-world passwords used in the experiment are selected from the Rockyou password dataset. As discussed in §V-C, the number of case switches and keyboard switches are two main factors that influence the ability to recover passwords. Therefore, we first categorize these passwords by the number of case switches and keyboard switches, then select passwords from different categories. Based on our analysis of the Rockyou dataset, we found that among all users using passwords containing at least one letter, passwords used by 99.65% of them have a case switch and keyboard switch count of no more than 2 times. Therefore, we focused our experiment on the 9 categories of passwords that fall within this range. Within each category, we sorted passwords that are longer than 8 characters and contain only English keyboard characters based on their popularity (i.e., the number of users), and selected the top 12 passwords for the experiment. In total, we selected 108 (9 12) real-world passwords. The selected passwords are listed in Table III.

Experiment Guidelines. The selected 108 passwords were randomly distributed to the 20 volunteers, with 8 volunteers being assigned 6 passwords and 12 volunteers being assigned 5 passwords. To simulate the real attack scenario, we assumed that different people use different passwords and therefore each password was only assigned to one volunteer. Furthermore, to simulate the fact that people are typically familiar with their own passwords, each password was entered 20 times during the experiment. To ensure that the volunteers were familiar with the assigned passwords, we heuristically omitted data collected during the first 10 inputs of a password and only considered the last 10 user inputs of each password as valid data for the experiment.

Metrics. We define three metrics for our experiment: $TPRCS$, $TPRNCS$ and one-time password recovery success rate. $TPRCS$ and $TPRNCS$ are previously defined in §V-C and are introduced to evaluate the effectiveness of the time-side-channel-based switch detection method. We define a password recovery process as successful if the real password is included in the set of recovered possible passwords. In real-world scenarios, attackers generally only have one chance to steal user passwords. Therefore, we propose to use the one-time password recovery success rate as a metric, which is defined as the ratio of successfully recovered passwords (out of the selected 108 passwords) in a single round of password input. Note that similar to the one-time password recovery success rate, $TPRCS$ and $TPRNCS$ are also independently calculated between rounds.

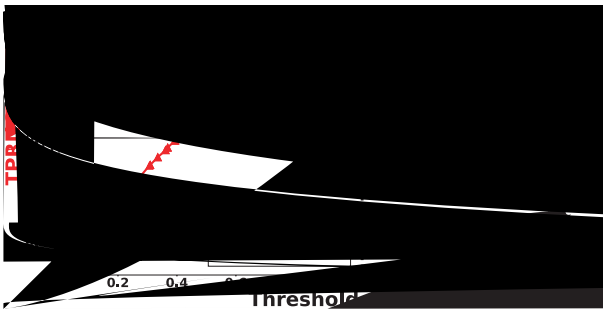
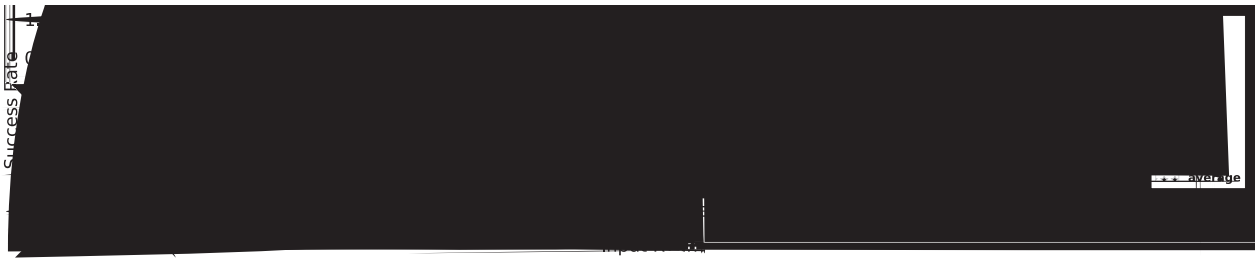
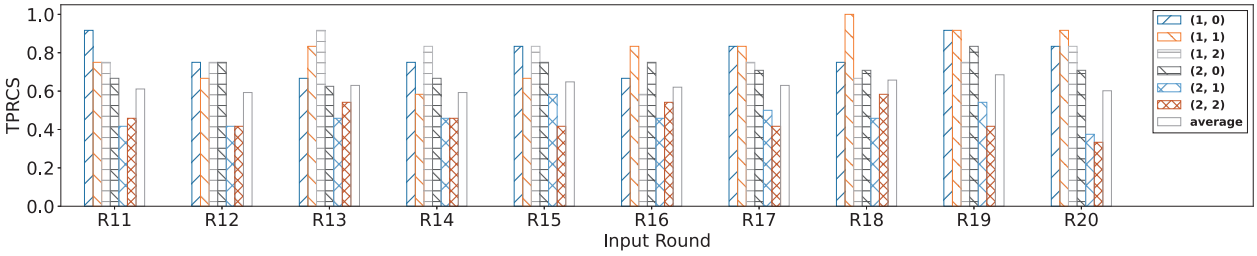


Fig. 4: Calculated $TPRNCS$-threshold and $TPRCS$-threshold curve on the MOBIKEYdataset

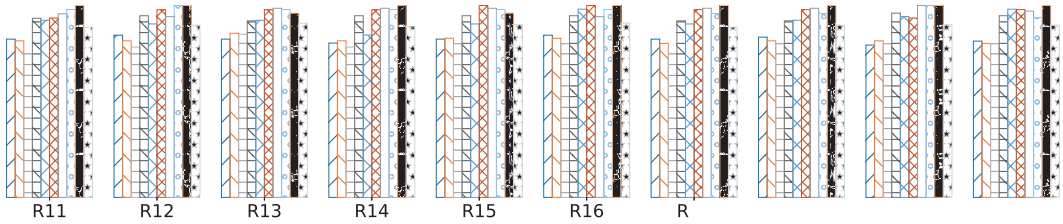
General Threshold Calculation. To calculate a general threshold for detecting case switches as discussed in §V-C, we



(a) One-time success rate



(b) TPRCS



(c) TPRNCS

Fig. 5: Experiment results of the CONQUER attack

conducted a preliminary experiment on the strong password (*:tie5Roanl*) dataset of the MOBIKEY keystroke dynamics password database [4]. This dataset includes 3303 records and provides many different features of keystroke dynamics when entering the password, including time intervals between keystrokes. Using the time intervals between keystrokes, we can collect the time intervals between entering two letters. As discussed in §V-C, the time interval between entering "e" and "5" was discarded during normalization. However, we did kept the time interval between entering "5" and "R" because "R" is the only uppercase letter in the password. This specific time interval was calculated as the summation of keystroke intervals between "abc" and "shift", and "shift" and "R", while the interval between "5" and "abc" was ignored to minimize the impact of keyboard switching. After that, the optimal threshold that maximizes $TPRCS$ $TPRNCS$ was calculated as previously discussed.

Figure 4 describes the relationship between $TPRNCS$ and the threshold, as well as the relationship between $TPRCS$ and the threshold on the MOBIKEYdataset. Using the metric mentioned above, the threshold was calculated to be 1.028, resulting in $TPRCS$ and $TPRNCS$ values of 0.9700 and 0.9692, respectively. To validate the effectiveness and generality of the time-and-model-based password recovery method, we adopted this optimal threshold (i.e., 1.028) derived from the MOBIKEYdataset throughout our experiment.

B. Experiment Results

RQ1. How does the pre-calculated threshold perform in detecting case switches?

We measure $TPRCS$ and $TPRNCS$ to answer this question. $TPRCS$ and $TPRNCS$ were calculated for the entire set of 108 selected passwords, as well as for each category of passwords, using the data collected from the last 10 rounds of password inputs. The threshold used to detect case switches was set to 1.028 as previously discussed. The results are shown in Figure 5(b) and Figure 5(c), where the notion "(*a*, *b*)" represents a category of passwords with *a* case switches and *b* keyboard switches. Note that for $TPRCS$, passwords in categories (0, 0), (0, 1), and (0, 2) do not have case switches and are therefore omitted in the figure. The overall $TPRCS$ and $TPRNCS$ within the last 10 rounds ranges from 59.26% to 68.51% and 88.69% to 90.87%, respectively. Interestingly, we observed two diametrically opposite trends for $TPRCS$ and $TPRNCS$. For $TPRCS$, the value decreases as the number of case switches increases, and the negative impact of the number of keyboard switches on $TPRCS$ becomes greater as the number of case switches increases. However, when it comes to $TPRNCS$, the value increases as the number of case switches increases, and the increase in the number of keyboard switches has a positive impact on this value. We believe that

this phenomenon occurs because the increase of these two numbers may make passwords more difficult to enter, thereby violating the common input pattern exploited by the proposed temporal-side-channel-based case switch detection method.

RQ2. Can CONQUER effectively steal user passwords exploiting the two side channels?

To answer this question, we first measure the one-time password recovery success rate. Similarly, the success rate was calculated for all 108 selected passwords and for each category of passwords within the last 10 rounds of input. Figure 5(a) shows the obtained one-time success rate. The overall one-time success rate ranges from 60.18% to 69.44%, with an average of 64.91%. For passwords in categories (0, 0), (0, 1) and (0, 2), the success rate is 100% because they follow the three most common patterns described in §V-C. For other categories of passwords, the one-time recovery success rate is related to $TPRCS$ and therefore follows the same pattern as $TPRCS$. The one-time recovery success rate is relatively low for passwords in categories (2, 1) and (2, 2) due to their high complexity. However, such cases are rare in real-world, because among all users who use passwords with at least one letter, only 0.02041% of them have deployed passwords in these two categories. Additionally, we did not observe the impact of smartphones keyboard layouts on our experiment of 20 subjects, as our method of detecting case switches utilizes normalized typing intervals between letters, which will not be affected by keyboard layouts. Finally, we evaluated the stealthiness of CONQUER. CONQUER does not necessitate any foreground UI operations, instead, it only executes background queries. Therefore, except for possible delays caused by the queries, victims should not detect any unusual behavior. Our experiment validated this point: we explicitly asked the volunteers if they had observed any anomalies (including delays), and none of them did. To demonstrate the high query efficiency of CONQUER, we randomly generated 20 passwords of length 16 and recorded the time it took to successfully query them. The results show that on average, it only takes 174ms to query a password of length 16, including the time for inter-process communication (IPC) and the time for actually processing the query on the victim side (see §VII-B). In summary, CONQUER is robust and can effectively recover passwords in most real-world scenarios.

RQ3. Which Android versions are affected and how many apps are subject to CONQUER attack?

Through manual verification, all Android versions from 4.1 to 12 (i.e., all currently officially supported versions) are subject to CONQUER, which means that all system-provided password field UI components and apps using these components are vulnerable. However, Android apps may use custom password input boxes, which could potentially avoid this vulnerability. As such, we have designed and implemented a framework for detecting vulnerable custom password input boxes used by Android apps. Our framework is built on top of Jadx [16] and Soot [30] to decompile Android apps, extract layout files, and perform static analysis. We look for self-defined elements whose android:inputType attributes represent password input boxes and check if they follow two rules based on our root cause analysis (see §VII-B) : (i) the class of the custom password input box must be a subclass of TextView . (ii) the class and its superclasses (excluding TextView) must not override the findViewsWithText() method, and either the getAccessibilityNodeProvider() method is not overridden by these superclasses or it is overridden but no class in the app has extended AccessibilityNodeProvider .

To measure the impact of CONQUER on these apps, we collected 324,125 Android apps from AndroZoo [3] and performed a large-scale security analysis on these apps using our framework. We were able to successfully test 324,080 apps, while 45 apps were unable to be tested because Jadx failed to decompile them. Out of the 324,080 apps that we successfully tested, 13,786 have custom password input boxes, and 13,001 out of the 13,786 apps (94.30%) implement their own custom password input boxes based on TextView . By applying our detection rules on these TextView -based custom password input boxes, we found that all of them (100%) are vulnerable to CONQUER. The evaluation results show that the vulnerability has not been previously recognized by the community and has a huge security impact.

VII. DISCUSSION

A. Responsible Disclosure and Ethical Considerations

We take ethics into the highest consideration. First, we responsibly disclosed our findings to Google. However, Google decided not to fix this vulnerability for two main reasons: 1) this behavior is required for the accessibility service to function

```
private void findAccessibilityNodeInfosByTextUiThr
↪ ead(Message message)
↪ {
  final int flags = message.arg1;
  ...
  ...
  final int accessibilityViewId = args.argi1;
  final int virtualDescendantId = args.argi2;
  ...
  List<AccessibilityNodeInfo> infos = null;
  try {
    ...
    final View root = findViewByAccessibilityId(ac
    ↪ cessibilityViewId);
    if (root != null && isShown(root)) {
      AccessibilityNodeProvider provider =
        root.getAccessibilityNodeProvider();
      if (provider != null) {
        infos = provider.findAccessibilityNodeInfo
        ↪ sByText(text,
        ↪ virtualDescendantId);
      } else if (virtualDescendantId ==
      ↪ AccessibilityNodeProvider.HOST_VIEW_ID) {
        ArrayList<View> foundViews =
        ↪ mTempArrayList;
        foundViews.clear();
        root.findViewsWithText(foundViews, text,
          View.FIND_VIEWS_WITH_TEXT
        | View.FIND_VIEWS_WITH_CONTENT_DESCRIPTION
        | View.FIND_VIEWS_WITH_ACCESSIBILITY_NODE_
        ↪ PROVIDERS);
        ...
      }
    }
  } finally {
      ...
  }
}
```

Listing 1: Client-side findAccessibilityNodeInfosByText() request handler in the Android source code

by an app with the accessibility service permission, the call is forwarded via IPC and the request is finally handled by the View object to be searched. A View object has two ways to handle the request: First, the method getAccessibilityProvider() is called. If an AccessibilityProvider object is returned, the request is finally handled by the findAccessibilityNodeInfosByText(text) method of the returned object. Second, if getAccessibilityProvider() returns null, the request is handled by the findViewsWithText() method of the View object itself.

By default, the getAccessibilityProvider() method in View will return null, and hence findViewsWithText() will be invoked to handle the request. However, the findViewsWithText() method in View only searches the given text in the content description of the View object and hence is not vulnerable to the proposed attack. TextView is a subclass of View. In TextView, the findViewsWithText() method is overridden to search the given text in the object's own text, while the getAccessibilityProvider() method is not overridden. Therefore, TextView and all Android-provided TextView-based classes, including those that are widely used as password input boxes (e.g., EditText), are vulnerable to CONQUER because they do not override the two methods.

C. Possible Mitigation

While CONQUER is hard to be defended as it is powered by the normal functionalities of the accessibility service, there are still possible mitigation measures. There are two possible ways to mitigate the attack based on the discussion in §VII-B: either at the system level, before the findAccessibilityNodeInfosByText(text) request is handled by the client, or at the application level, after the request is dispatched to the client.

> System level mitigation There are several ways to fix this vulnerability at the system level. One option is to enforce security checks inside the server-side accessibility service API findAccessibilityNodeInfosByText(text) to ensure that a password node is not allowed to be searched. However, identifying password nodes could be challenging in general due to the existence of custom password input boxes. Another way is to make the API only search the given text inside content descriptions but not the contained texts. However, this approach may hinder the functionalities of the accessibility service.
>
> Application level mitigation To mitigate this vulnerability, Android app developers should always adopt custom password fields instead of system-provided TextView-based classes as password fields. Custom password fields inherited directly or indirectly from TextView should either override the findViewsWithText() method to make sure the password text is not searched, or override the getAccessibilityNodeProvider() method to return a custom AccessibilityNodeProvider-based object.

D. Limitations

Our CONQUER is not perfect, and it has the following limitations. First, though the range of possible passwords is greatly reduced to make the attack more practical, the remaining number of possible passwords could still be too large to perform a successful login attempt when there is a limitation on failed login attempts. Second, the time-and-model-based password recovery method is not reliable for long or complex passwords due to accidental input errors or different typing habits (e.g., using keyboard pop-ups to choose characters). However, smartphone sensors can accurately detect touch events (e.g., [36]). Future research can use sensors to more accurately distinguish case changes based on the number of touches between letters. Additionally, passwords may have semantic features [31], [33]. Future research can also apply these semantic patterns to recover passwords. Third, if victims use password managers to automatically fill in their passwords, CONQUER can only steal case-insensitive passwords by actively issuing content queries. As discussed in §VI-B, querying passwords is efficient and if the victim does

not press the login button within a very short period after the password is filled in, the case-insensitive password can still be obtained. However, original passwords cannot be efficiently recovered due to the lack of input timing information. Previous research has shown that the use of password managers is not common, particularly on mobile phones [27], [2].

VIII. Related Work

Android Accessibility Service Abuses. The Android accessibility service has been proven to have exploitable design shortcomings by previous research. Kraunelis *et al.* [21] demonstrated that the Android accessibility service can be exploited by malware to perform malicious actions such as gaining control of the screen and stealing user credentials through phishing. Jang *et al.* [17] studied the security of accessibility support on four popular platforms and identified several vulnerabilities. Fratantonio *et al.* [11] proposed the famous "cloak and dagger" attack exploiting both overlay and the accessibility service. Interestingly, the "cloak and dagger" attack can be executed even without the overlay permission before Android 8.0 [37]. Aonzo *et al.* [5] showed that it is possible to conduct phishing attacks against password managers by exploiting the accessibility service. Kalysch *et al.* [18] discovered several security flaws in the accessibility service and discussed corresponding countermeasures. Diao *et al.* [9] conducted a systematic study of the Android accessibility framework through code review and app scanning, and discussed several shortcomings as well as corresponding attacks exploiting these weaknesses. Evidence has proven that various real-world malware [10], [1], [39] have exploited the attacks mentioned above. However, none of these works have discovered the query-based side channel in the Android accessibility service.

Accessibility Service Assisted Password Stealing Attacks. Kraunelis [21] *et al.* pointed out that the accessibility service can be exploited to steal user passwords through phishing, but the malware has to completely disguise itself as a benign app, which could be hard for complex closed-source apps. Jang *et al.* [17] demonstrated that an Android malware exploiting the accessibility service can alter system settings programmatically without user consent and register a malicious text-to-speech (TTS) application to steal passwords. However, this approach no longer works on newer versions of Android. Instead, a series of UI operations are needed to accomplish the same goal. Fratantonio *et al.* [11] proposed three password stealing attacks, two of which require the assistance of overlay. This results in the display of an alert window on Android 8.0 or later, which makes it less practical these days. The other attack that only uses the accessibility service is achieved by inferring keystrokes from keyboards, but this requires the keyboard used by the victim to be vulnerable. However, some keyboards have have addressed this vulnerability [18]. Kalysch *et al.* [18] found that by exploiting screen recording or accessibility events sniffing, the most recently entered password character can be acquired, as the character is displayed on the screen for a short period of time. However, this can be easily defended by turning off the "Make passwords visible" option in Settings . Therefore, all existing accessibility-service-assisted password stealing attacks are much less practical or feasible nowadays than when they were first proposed. In contrast, our proposed attack is more stealthy, general, and practical.

Android Accessibility Service Defenses. Compared with attacks against the accessibility service, defenses on the framework are rarely focused. Naseri *et al.* [25] proposed a framework to help developers automatically detect and fix Android apps that may leak passwords through the accessibility service. Huang *et al.* [15] recently proposed a privacy-enhanced accessibility framework to strike a balance between the regular functionality of the accessibility framework and its security mechanisms. While this is not the focus of this work, more comprehensive security mechanisms should be studied in the future.

Side-Channel-based Keystroke Inference. Previous research has studied the feasibility of inferring keystrokes through various side channels. The temporal side channel is one of the most commonly exploited methods for keystroke inference [29], [6]. Smartphone sensors are also used to infer user keystrokes [38], [32], [19], [36], [24], [22] due to the vast amount of information they provide. We also exploit the temporal side channel, but unlike previous works, it is used to detect case switches rather than inferring keystrokes.

IX. Conclusion

In this work, we propose CONQUER: a novel content query assisted password stealing attack. CONQUER breaks existing Android defenses against password stealing attacks by exploiting a query-based side channel in the Android accessibility service, and can be abused to launch password stealing attacks in real-world scenarios. To make CONQUER practical, we introduce the *lazy query* technique to disambiguate query results, the *active query* technique to determine query timing, and the temporal side channel and state machine to recover case-sensitive passwords. Our experiment shows that CONQUER can steal user passwords with a high success rate. The attack has affected all Android versions from 4.1 and 12 and many Android apps. CONQUER has not been recognized by the community and poses a significant security risk.

Acknowledgment

We would like to thank Kang Jia and the anonymous reviewers for their valuable feedback on this work. This research was supported in part by National Natural Science Foundation of China Grant Nos. 62022024, 61972088, 62072103, 62102084, 62072102, 62072098, 62232004, and 61972083, by US National Science Foundation (NSF) Awards 1931871, 1915780, and US Department of Energy (DOE) Award DE-EE0009152, Jiangsu Provincial Natural Science Foundation of China Grant No. BK20190340, Jiangsu Provincial Key R&D Program (Nos. BE2021729, BE2022680 and BE2022065-4), Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant Nos. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] 0x1c3n, "Anubis android malware analysis," 2021. [Online]. Available: https://0x1c3n.tech/anubis-android-malware-analysis

[2] N. Alkaldi and K. Renaud, "Why do people adopt, or reject, smartphone password managers?" in *Proceedings of the 1st European Workshop on Usable Security (EuroUSEC)*, 2016.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of Android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, 2016, pp. 468–471.

[4] M. Antal and L. Nemes, "The MOBIKEY keystroke dynamics password database: Benchmark results," in *Software Engineering Perspectives and Application in Intelligent Systems: Proceedings of the 5th Computer Science Online Conference (CSOC)*, R. Silhavy, R. Senkerik, Z. K. Oplatkova, P. Silhavy, and Z. Prokopova, Eds. Cham: Springer International Publishing, 2016, pp. 35–46.

[5] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, "Phishing attacks on modern Android," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1788–1801.

[6] L. Cai and H. Chen, "TouchLogger: Inferring keystrokes on touch screen from smartphone motion," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec)*, 2011.

[7] W. Cao, C. Xia, S. T. Peddinti, D. Lie, N. Taft, and L. M. Austin, "A large scale study of user behavior, expectations and engagement with Android permissions," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 803–820.

[8] E. Cebuc, "How are we doing with Android's overlay attacks in 2020?" 2020. [Online]. Available: https://labs.f-secure.com/blog/how-are-we-doing-with-androids-overlay-attacks-in-2020/

[9] W. Diao, Y. Zhang, L. Zhang, Z. Li, F. Xu, X. Pan, X. Liu, J. Weng, K. Zhang, and X. Wang, "Kindness is a risky business: On the usage of the accessibility APIs in Android," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019, pp. 261–275.

[10] T. Fabric, "The rage of Android banking trojans," 2021. [Online]. Available: https://www.threatfabric.com/blogs/the-rage-of-android-banking-trojans.html

[11] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: From two permissions to complete control of the UI feedback loop," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017, pp. 1041–1057.

[12] Google, "Create your own accessibility service," 2021. [Online]. Available: https://developer.android.com/guide/topics/ui/accessibility/service

[13] Google, "Accessibilitynodeinfo," 2022. [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo

[14] Google, "Talkback," 2022. [Online]. Available: https://support.google.c

[38] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 373–382.

[39] E. Y. Şahin, "When your phone gets sick: Flubot abuses accessibility features to steal data," 2021. [Online]. Available: https://www.srlabs.de/bites/flubot-abuses-accessibility-features-to-steal-data