# Boosting the Guessing Attack Performance on Android Lock Patterns with Smudge Attacks

Seunghun Cha[1], Sungsu Kwag[1], Hyoungshick Kim[1] and Jun Ho Huh[2]
[1]Department of Software, Sungkyunkwan University, Republic of Korea
[2]Honeywell ACS Labs, Golden Valley, MN USA
{sh.cha, kssu1994, hyoung}@skku.edu
junho.huh@honeywell.com

## ABSTRACT

Android allows 20 consecutive fail attempts on unlocking a device. This makes it difficult for pure guessing attacks to crack user patterns on a stolen device before it permanently locks itself. We investigate the effectiveness of combining Markov model-based guessing attacks with smudge attacks on unlocking Android devices within 20 attempts. Detected smudges are used to pre-compute all the possible segments and patterns, significantly reducing the pattern space that needs to be brute-forced. Our Markov-model was trained using 70% of a real-world pattern dataset that consists of 312 patterns. We recruited 12 participants to draw the remaining 30% on Samsung Galaxy S4, and used smudges they left behind to analyze the performance of the combined attack.

Our results show that this combined method can significantly improve the performance of pure guessing attacks, cracking 74.17% of patterns compared to just 13.33% when the Markov model-based guessing attack was performed alone—those results were collected from a naive usage scenario where the participants were merely asked to unlock a given device. Even under a more complex scenario that asked the participants to use the Facebook app for a few minutes—obscuring smudges were added as a result—our combined attack, at 31.94%, still outperformed the pure guessing attack at 13.33%. Obscuring smudges can significantly affect the performance of smudge-based attacks. Based on this finding, we recommend that a mitigation technique should be designed to help users add obscurity, e.g., by asking users to draw a second random pattern upon unlocking a device.

## Keywords

Pattern Lock; Guessing Attack; Smudge Attack

## 1. INTRODUCTION

To help smartphone users select memorable and secure authentication secrets, in 2008, Google introduced a graphical password scheme (referred to as "Android pattern lock" or "Android screen lock pattern") adopted from "Pass-Go" [20] for Android devices, which asks users to create and remember a graphical pattern on a $3 \times 3$ grid. This scheme has quickly emerged as the most popular locking method for Android devices [22]. Many users perceive patterns as quicker and less error-prone unlocking method than PIN [23]. It is unclear, however, whether their security is guaranteed in practice. Several studies [19, 21] demonstrated that the space of real patterns might be much smaller than the theoretical space, making password guessing attacks feasible.

To mitigate guessing attacks, Android only allows up to 20 consecutive fail unlock attempts—after 20 consecutive fail attempts, Android displays the "Too many pattern attempts" error message, and asks the user to log in with a Google account to unlock the device. This policy is effective against online guessing attacks, but might not be sufficient to prevent a well-known side channel attack called *smudge attack* [5] that uses fingerprint smudges left behind on a touchscreen to infer a correct pattern.

Drawing a pattern with an oily finger leaves smudges on a touchscreen. Such smudges can provide useful information for efficiently guessing a pattern. Aviv et al. [5] examined the feasibility of this smudge-based inference attack on the Android lock pattern by testing various experimental conditions (e.g., lighting and camera angles) under which smudge-based inference attacks can easily be performed. Zezschwitz et al. [24] also showed that the Android lock pattern was vulnerable to smudge attacks through a lab experiment. Their results indicated that smudge attacks can be effective in cracking user patterns. However, their security analysis was mainly based on participants' self-reported ratings on the possibility of correctly guessing patterns from looking at smudges. To the best of our knowledge, there is no previous work that has actually implemented a fully working smudge attack (or guessing attack) tool and tested its real performance.

We propose a novel "smudge-supported pattern guessing" (smug) attack that pre-computes all the possible patterns using detected smudges, significantly reducing the pattern space that needs to be brute-forced with a guessing attack. To improve practicality of smudge attacks, we used image processing techniques to automatically detect smudges from a picture of an Android device. Detected smudges are used to generate a list of all possible patterns, and guessing attack is performed within that small pattern space.

To evaluate the effectiveness of smug attack, we first constructed an $n$-gram Markov model with 219 (70%) of 312 real-world patterns collected through an Android app (only under users' agreement). Next, we recruited 12 participants, and asked each participant to draw 30 patterns randomly selected from the remaining 93 (30%) patterns. Finally, we performed smug attack using the smudges they left behind. When we asked the participants to merely unlock a given device, our results showed that smug attacks can significantly outperform pure guessing attacks, cracking 74.17% of 360 ($= 12 \times 30$) patterns within 20 unlock attempts com-



*ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates*

DOI: http://dx.doi.org/10.1145/3052973.3052989

pared to just 13.33% being cracked when Markov-based guessing attacks were performed alone. To closely resemble a real-life phone usage scenario, we also asked them to use the Facebook app for a few minutes after unlocking a device. Smug attacks still managed to crack 31.94% of those 360 patterns compared to just 13.33% being cracked under pure guessing attacks. Hence, we recommend that a mitigation technique should be designed to help users add more smudge obscurity by, e.g., drawing a second random pattern. Our key contributions are summarized as follows:

1. We proposed the combined smug attack, and implemented the first *fully automated and working* tool that is capable of performing both smudge and guessing attacks. Using our tool, 20 possible pattern candidates with high likelihood can be identified automatically from a given touchscreen image (that contains smudges), taking about 18 seconds on average. This is a significant advancement from previous literature that merely speculated the likely performance of smudge attacks based on user feedback, and without a real implementation of smudge attacks or guessing attacks.

2. Using the smug attack tool and real-world pattern datasets, we evaluated the performance of smudge attacks, dictionary-based guessing attacks, and the combined smug attacks. We used the smudges left behind from the participants who were asked to perform real-world phone usage scenarios such as using the Facebook app for a few minutes. Our results suggest that smug attacks (with 74.17% attack success rate) significantly outperformed dictionary-based pure guessing attacks (13.33%). Even when obscuring smudges were added under the Facebook usage scenario, smug attacks still showed a higher attack success rate (31.94%) compared to pure guessing attacks.

3. In contrast to inconclusive findings from previous literature, we also identified limitations of smudge-based attacks through full implementation and testing them on real-world patterns, demonstrating that obscuring smudges can significantly downgrade the performance of smudge-based attacks.

4. We explored potential countermeasures to mitigate smudge attacks and particularly evaluated an obscurity-based mitigation technique that helps users to add effective obscuring smudges, showing that it can significantly reduce the performance of smug attacks from 74.17% to 34.44%. Unlike existing smudge attack mitigation schemes (e.g., [17]), our recommendation does not require any changes in using an Android screen lock pattern.

The rest of the paper is structured as follows. Section 2 explains Android screen lock patterns and attack model. Section 3 describes smug attack in detail. Section 4 explains how real-world patterns were collected. Attack optimization techniques are covered in Section 5, and attack performance is discussed in Section 6. Mitigation techniques are discussed in Section 7. We discuss attack limitations in Section 8. Related work is covered in Section 9, and our conclusions are in Section 10.

# 2. BACKGROUND

## 2.1 Android screen lock patterns

Android screen lock pattern is one of the most popularly used graphical password schemes [2]. A user is asked to choose a *secret* pattern consisting of consecutive segments (lines connecting points) on a $3 \times 3$ grid, and in the authentication phase, the user has to draw that pattern on the grid to unlock the user's Android device (see Appendix A). For notational convenience, the following conventions are adopted throughout the paper. The 9 points on the grid are numbered from 1, starting with the point on the top left corner, to 9, which is the point on the bottom right corner of the grid. A "segment" in a pattern is defined as a line that connects two points together. An Android pattern must consist of at least four points, and a point cannot be used more than once.

In theory, the total number of all possible patterns is 389,112 ($\approx 2^{18}$), which is much larger than the password space of 10,000 four-digits PINs that are also commonly used to lock phones. Despite this relatively larger password space, users still choose weak patterns that are susceptible to various attacks like guessing attacks [19, 21], smudge attacks [3, 5], sensor-based side channel attacks [6], and shoulder surfing attacks [25]. This paper focuses on evaluating the effectiveness of smudge attacks and guessing attacks based on real-world datasets and fully automated implementation.

## 2.2 Attack model and assumptions

This section describes our threat model and assumptions. People often use oily fingers to perform various tasks on their phones, leaving smudges behind on the touchscreen. Some common tasks include unlocking phones by drawing a pattern, sending texts, surfing the Internet, playing games, and so on. Oily smudges left behind from multiple tasks would obscure the actual smudge traces that need to be collected to guess the right screen lock pattern. Given those challenges, an attacker's goal is to steal an Android phone from someone with a high profile (e.g., a celebrity or politician), use a smart smudge attack to quickly unlock the stolen phone within 20 attempts, and access his or her confidential data.

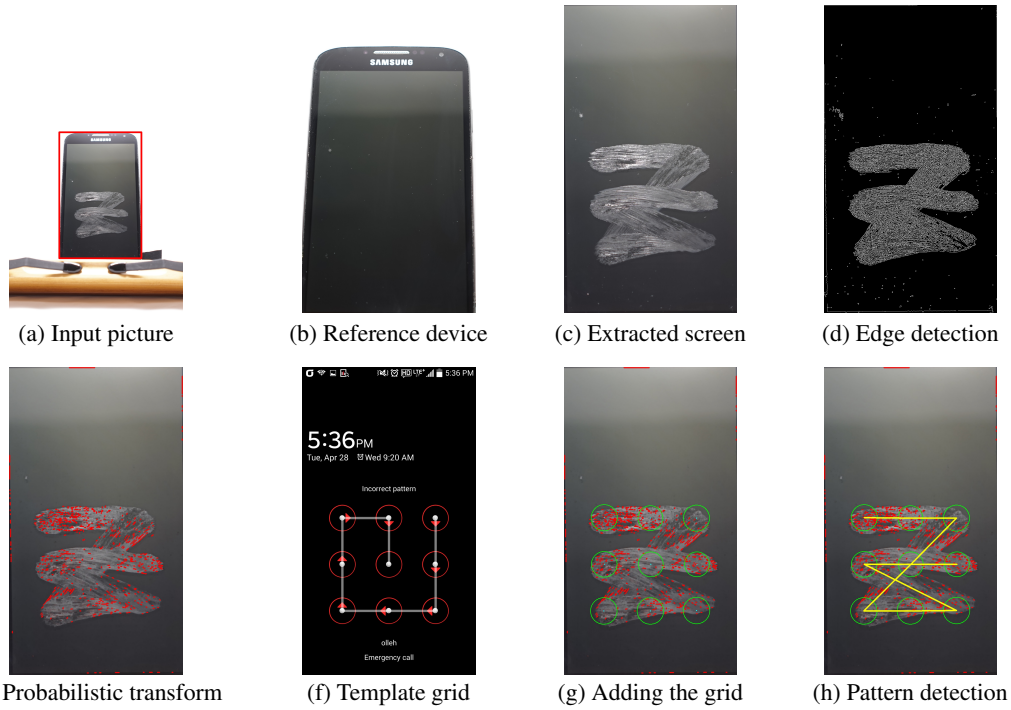
Such an attack scenario is becoming popular, and more and more mobile phone manufacturers are enabling full disk encryption on their devices to protect user data from strangers and hackers. FBI's recent attempt to unlock an iPhone owned by a terrorist is an example of this scenario [9]. According to a survey conducted in London [8], more than 60,000 mobile devices were left in the back of taxis during a six month period. This number indicates that a large number of lost mobile devices could potentially become a target for smudge attacks and guessing attacks.

The effectiveness of this attack depends on the amount and clarity of smudges remaining on the stolen phone, and how much information about screen lock patterns is contained in the smudges left behind. In performing such an attack, we assume that (1) the attacker is in possession of the victim's phone for a few minutes, (2) the phone has sufficient amount of smudges left behind, and (3) the remaining smudges contain some hint about the actual unlock pattern. Those three assumptions are integral when it comes to implementing a smudge-based attack.

We show that such assumptions may often be reasonable through the user studies for simulating popular phone usage scenarios presented in Section 5 and 6.

# 3. SMUDGE-SUPPORTED PATTERN GUESSING ATTACK

The proposed smudge-supported pattern guessing (smug) attack combines two techniques: (1) image processing to infer possible patterns from smudges, and (2) sorting patterns based on the occurrence probabilities computed using an $n$-gram Markov model, which could be constructed using real-world pattern datasets. When an attacker feeds in the picture containing Android device's screen to the smug attack tool, it automatically analyzes smudges, creates




(a) Input picture (b) Reference device (c) Extracted screen (d) Edge detection

(e) Probabilistic transform (f) Template grid (g) Adding the grid (h) Pattern detection

**Figure 1: Overall process for recovering the user's pattern drawing with its smudges.**

segments, and spits out possible patterns. The number of possible patterns will depend on the clarity and representatives of smudges. Since Android only allows 20 failed attempts, there is a need of another smarter mechanism to try out the possible patterns. To that end, we use an $n$-gram Markov model to sort possible patterns in descending order, starting from the pattern with the highest occurrence probability. The attack is successful if the correct pattern is found within 20 guessing attempts and the Android device is unlocked.

Smug attack involves the following four steps: (i) extracting the exact touchscreen area from a picture of a target device; (ii) identifying pattern-relevant smudge objects from the extracted pattern input area; (iii) generating possible pattern segments from the identified smudge objects; (iv) generating possible pattern candidates, and ordering them in a descending order according to their occurrence probabilities. The last step allows the attack to try the most likely occurring patterns first. As for image processing, we used OpenCV [1], a popular open source computer vision library, to quickly implement the basic operations used in our smug attack tool. Each step is described in detail in the following sections.

## 3.1 Extracting the pattern input area

The obvious first step of smug attack is to take a picture of a device using an optimal camera and light setting. Our recommended camera and light setting is described in Section 5.2. Inherently, the setting can be changed depending on the target device. Figure 1(a) to (c) show the processes involved in obtaining the exact touchscreen area from a given picture of an Android device.

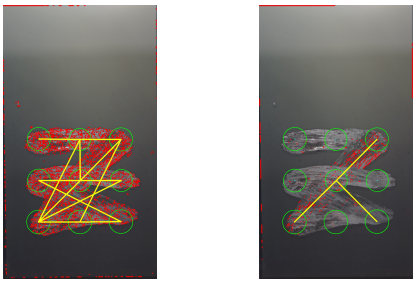
First, given a picture of a mobile device (e.g., as shown in Figure 1(a)), we use an image matching algorithm with reference device images, such as the Samsung Galaxy S4 image (see Figure 1(b)), to recognize the device (see the red rectangle in Figure 1(a)). The most similar reference device image is automatically selected from a pre-stored set of reference images by measuring the similarities between images. Once the device object is identified using a matching reference image, the touchscreen part is automatically cropped and adjusted using a perspective transform technique to tilt the touchscreen 60 degrees to the left. Then, the touchscreen is scaled to a predefined image size (e.g., 810 × 1440 pixels). This scaled image is then compared against the reference image to locate the x and y coordinates of the $3 \times 3$ grid objects. As a result, a "bird's eye view" image of the touchscreen is created as shown in Figure 1(c).

## 3.2 Identifying smudge objects

In our implementation, the target touchscreen image (i.e., Figure 1(c)) is first binarized to enhance the visibility of smudges of interest. Canny edge detection [10] is applied to locate the regions where fingers have touched the screen (see Figure 1(d)). Located regions are then processed using the probabilistic Hough transformation [15] to extract the edges of interest (see the red edges in Figure 1(e)). To locate the exact pattern input area (i.e., where the pattern-relevant smudges are), we also use a reference image with the $3 \times 3$ grid (see Figure 1(f))—the center point and radius of each circle object on the grid can be calculated from this reference image by using the Hough circle transform [7]. The computed $3 \times 3$ grid objects can be incorporated into the captured touchscreen image with smudges (see Figure 1(g)). Finally, we apply our own heuristic algorithm with the detected red edges to decide whether there exists a segment between two grid points (see Section 3.3). Figure 1(h) shows an example of detected segments (yellow lines) inferred through those processes. Using those detected segments, a probabilistic password model, such as an $n$-gram Markov model, can identify possible and likely pattern candidates.

For the Canny edge detection algorithm, we set the low threshold value to 10 and the high threshold value to 30. For the probabilistic Hough transformation, we set the minimum line length to 2 pixels, the maximum gap to 5 pixels, and the threshold value to 10. It is important to set appropriate parameter values for filtering valid edges

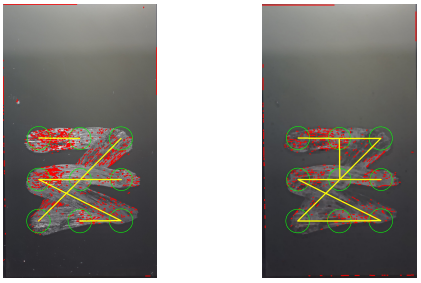
that are actually relevant to the correct lock pattern. For example, in the probabilistic Hough transformation, if a threshold value is too low for edge pixel's gradient value, we may end up with too many incorrect/false edges (caused by noise); if a threshold value is too high, we might miss a few real/true edges relevant to the correct pattern. Figure 2 shows the effects of threshold values in the probabilistic Hough transformation. Those parameter values were determined experimentally with a small number of test samples.



(a) Threshold = 2 (b) Threshold = 200

**Figure 2: Effects of threshold values in the probabilistic Hough transformation.**

Before Canny edge detection is complete, several morphological filters [13] can also be applied to remove short and isolated edges that appear due to noise. We tested several morphological operators such as dilation, opening, closing, and morphological gradient, but the morphological transformation with one-time erosion operation only works well for our application. Figure 3 shows the effects of applying erosion morphological transformation operations.



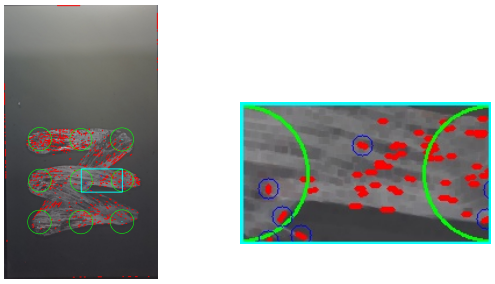
(a) No erosion (b) Two-times erosion

**Figure 3: Effects of erosion morphological transformation.**

After the probabilistic Hough transformation is performed, we only select the edges with a similar direction to the segment between two grid points to remove as many noisy edges as possible. If the angle between an edge and a pattern segment is less than or equal to about 13 degrees then we assume that they have a similar direction. Figure 4 shows what kinds of edges were filtered out. In Figure 4(b), the area between points 5 and 6 is scaled and cropped for improved visualization. To improve the accuracy of pattern segment decisions, we ignore several edges with a direction different to the direction of the segment between points 5 and 6 (see red edges in blue circles in Figure 4(b)). Smudges left behind due to the user's real pattern drawing actions might have a similar direction as the pattern segments.

## 3.3 Generating a set of segments forming the target pattern

Given the detected edges relevant to smudges, we need to generate a set of pattern segments which might be part of the correct



(a) Overall pattern (b) Between points 5 and 6

**Figure 4: Removal of noisy edges that move in a direction different to a pattern segment (highlighted in blue circles).**

pattern. To achieve this goal, we developed a heuristic algorithm with the detected edges shown as the red lines in Figure 1(g) to decide whether there exists a segment between two grid points, which is included in the user's pattern.

Our key idea is to (i) create an imaginary box between two grid points (i.e., the cyan box between points 5 and 6 as shown in Figure 4), (ii) count the number of the detected red edges within the box, and (iii) check whether that number is greater than or equal to a threshold value. In Section 5, we will discuss how to choose a proper threshold value.

In order to cover the overlapping screen lock trajectory, we considered any pair of two grid points that were either adjacently located or not adjacently located. Thus, our tool can also generate patterns (e.g., "2314") with an overlapping screen lock trajectory as well.

## 3.4 Enumerating pattern candidates

Given a set of detected segments, the final step of smug attack is to generate possible pattern candidates with those segments, and sort them in descending order of their occurrence likelihood. Intuitively, without any information about a victim's actual pattern, an attacker's optimal guessing strategy is to start with the most likely occurring patterns first.

Provided that the attacker has access to a sufficiently large set of real-world patterns (e.g., through a pattern database), an $n$-gram Markov model could be used to effectively compute occurrence likelihood probability of the pattern candidates identified.

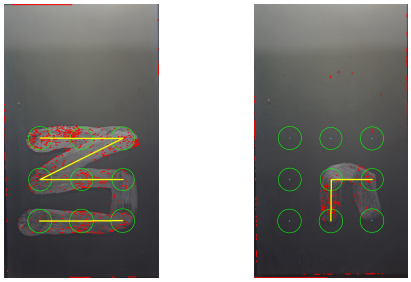
In our $n$-gram Markov model, we treat points in a pattern as events: since each point in a pattern represents a number between 1 and 9, a pattern can be represented as a sequence of numbers. The $n$-gram Markov model is used to estimate the probability of each number/point sequence $x_1, \cdots, x_m$ as

$$P_n(x_1, ..., x_m) = P(x_1, ..., x_{n-1}) \cdot \prod_{i=n}^{m} P(x_i | x_{i-n+1}, ..., x_{i-1})$$

In theory, when an $n$-gram Markov model is being constructed, it is best to use the highest possible $n$ given the size of the training dataset available to learn about the probabilities of events.

If there is not enough training data available, many $n$-gram occurrences will never be observed. Although a smoothing technique can be used to forcibly remove zero probability of such unseen events, this technique would eventually affect accuracy of computed probabilities. Through the analysis of experimental results in Section 5, we discuss an optimal $n$ value and smoothing technique for effectively using an $n$-gram Markov model in smug attack.

To improve guessing efficiency, we first sort the pattern candidates in descending order of the pattern length. This is based on



(a) Separated (b) Incompleted

**Figure 5: Undetected segments resulting in disjointed segment chunks in (a), and pattern length that is shorter than 4 in (b).**

intuition that longer patterns will comprise of more smudge objects, and have higher chance of being the correct pattern. Within this sorted list, for each pattern length, we sort again in descending order of the occurrence probabilities computed using an $n$-gram Markov model. This process can be explained using the example in Figure 1(h). In the case where the set of detected segments is $\{(1,2),(2,3),(3,5),(4,5),(4,9),(5,6),(5,7),(7,8),(8,9)\}$, the number of all possible Android patterns is 180. Smug attack will try the longest pattern that has the highest occurrence probability, which, in this case, is pattern "123578946." If this is not the correct pattern, smug attack will try other patterns sequentially until the target device is unlocked.

During the process of detecting pattern segments, however, we could miss valid segments that are included in the correct pattern (see the examples in Figure 5).

If that happens, we will inherently fail to guess the correct pattern because at least one valid segment will be missed. Missing segments could result in a disconnection with the detected segments or the number of detected segments being too small to try a valid pattern. To avoid such situations, a minimal number of connecting segments are added on to connect the disjointed segments so that valid Android patterns can be inferred. To find the minimal number of connecting segments, we simply brute-force possible segments until the segment chunks are connected. For example, in Figure 5(a), there are two disconnected chunks, "123456" and "789", which consist of the yellow lines. One additional segment can connect the two chunks and make the attack feasible. Smug attack adds this one additional connecting segment, and considers all possible pattern combinations consisting of the originally detected segments as well as the newly added connecting segment. Such cases were frequently observed in our experiments but our heuristics performed well in most cases. In the worst case scenario, if no segment is detected with smudges, we can simply perform the Markov model-based guessing attack on its own.

# 4. DATA COLLECTION

This section explains how we collected real-world Android patterns that have been used in evaluating the smug attack performance.

## 4.1 Collecting real-world patterns through Private Notes

One of the problems with designing an experiment that requires participants to draw their own patterns is that participants may decide not to draw their real patterns, and this could negatively affect the ecological validity of the experiment. To avoid that and minimize participants' risks associated with revealing their real patterns, we developed an Android app called Private Notes (see Appendix B) and made it available on Google Play to collect real-world pattern data. Because the Private Notes's lock pattern user interface is similar to the Android's default unlock user interface and serves a similar purpose, we claim that the collected pattern dataset closely resembles real-world Android lock patterns. Our study participants were then asked to just redraw those patterns to unlock given Android devices.

It was not our intention to collect any personal information. Only fully anonymized pattern data were collected under app users' agreement. When "Private Notes" is installed and launched for the first time, it asks for users' consent to anonymously disclose information about their pattern behavior for academic research purposes. Only when users agree, they are asked to create a new pattern to enable authentication and prevent unauthorized accesses to users' personal notes. After creating a pattern, users are asked to enter the pattern again for confirmation; if the re-entered pattern matches the original pattern, the pattern is saved; otherwise, users have to repeat the process until the two patterns match. We collected 312 patterns in total. From those patterns, about 70% of the collected patterns (219 patterns) were randomly selected and used as the training set to construct the $n$-gram Markov model described in Section 3. The remaining 30% of the patterns (93 patterns) were used as the testing set in optimizing smug attack parameters and evaluating the smug attack performance.

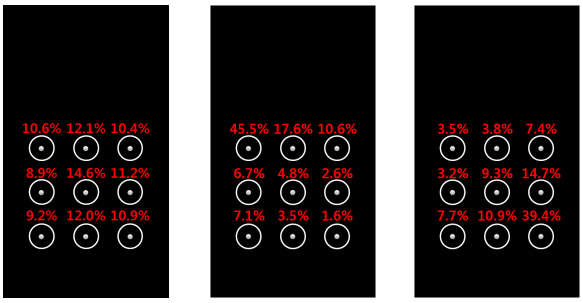
Users' security risks associated with sharing their patterns are much smaller than that of sharing passwords since most patterns are only ever used to unlock Android devices, and without physical access to users' devices, the harm that can be done with those collected patterns is limited. Such ethical perspectives of our research were carefully reviewed and approved by an Institutional Review Board (IRB) at a university.

## 4.2 Characteristics of real-world patterns

This section describes the statistical characteristics of the collected real-world patterns.

### *4.2.1 Frequency of the 9 points used in the collected patterns*

First, we analyze the usage frequency of each of the 9 points in the $3 \times 3$ grid. Those 9 points are numbered from 1, starting with the point in the top left corner, to 9, which is the point in the bottom right corner of the grid. The results are shown in Figure 6.




(a) 9 points (b) Start points (c) End points

**Figure 6: Frequency of each of the 9 points used in the collected patterns.**

In Figure 6(a), the most frequently used point is 5, which was used 266 times (14.6%). The least frequently used point is 4, which was only used 162 times (8.9%).

We also looked at preferred starting and ending points, respectively (see Figure 6(b) and (c)). The most frequently used starting point is 1 (used 142 times; 45.5%), and the least frequently used starting point is 9 (used 5 times; 1.6%). Points 6 (used 8 times; 2.6%) and 8 (used 11 times; 3.5%) were rarely used as starting points. The most frequently used ending point is 9 (used 123 times; 39.4%), and the least frequently used ending point is 4 (used 10 times; 3.2%). Overall, the usage frequencies across those 9 points were not evenly distributed.

#### 4.2.2 *Segments used*

A *segment* in a pattern is defined as a line that connects two points together. We counted the usage frequency of all of the segments used in the collected patterns. Figure 7 shows the proportion of the usage frequency for each segment: darker the color, higher the number of segments used.

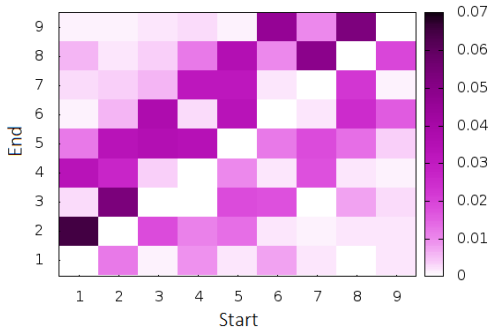



**Figure 7: Frequency of each of the segments used in the collected patterns.**

The total number of segments used is 1,511. But there are only 70 distinct segments in that 1,511. The most frequently used segments was (1, 2) which was used 97 times (6.42%). There are unused segments such as (4, 3) and (8, 1), which form long diagonal lines. We can also see two darker diagonal patterns from the lower left to the upper right, which implies that segments were usually chosen between geometric neighboring points. The usage frequency of segments appears to be biased towards those segments. Interestingly, directions of segments are also selectively chosen. Users seem to prefer segments that move from left to right, (1, 2), (2, 3), (4, 5), (5, 6), (7, 8), and (8, 9), and segments that move from top to bottom, (1, 4), (4, 7), (2, 5), (5, 8), (3, 6), and (6, 9).

Computing Shannon entropy [18] on those segments showed that the segment frequency distribution of real-world patterns has an entropy of 5.326. This is significantly lower than the entropy of equally used segment distribution, which is 6.340.

#### 4.2.3 *Pattern length*

Android patterns can have lengths between 4 and 9 (patterns must contain at least four points in Android). Our collected patterns have an average length of 5.830 with a standard deviation of 1.776. Short patterns are dominant, where the most commonly used pattern lengths are 4 and 5. However, patterns with length 9 are also quite common (see Figure 8). We surmise that this may be due to people using up all 9 points deliberately as a way for them to create patterns that are easy to recall.

The analyses presented in this section show that real-world patterns have skewed distribution in usages of points, segments, and pattern length. Such characteristics can be learned and exploited by guessing attacks, e.g., our Markov model-based attack, to compute likelihood of points and segments in advance, and make more efficient guesses.

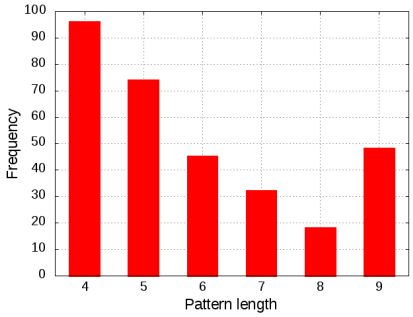



**Figure 8: Frequency of pattern lengths.**

## 5. FIRST RESULTS: SMUG ATTACK OPTIMIZATION

This section discusses several parameter choices for smug attacks, and recommends an optimal set of parameters to be used based on experimental results.

### 5.1 Methodology

For the first experiment, we recruited one male undergraduate student from the computer science department who currently uses an Android device to optimize the smug attack. We asked him to normally draw the 93 patterns in a given set of test patterns which are different from the 219 patterns used for constructing the $n$-gram Markov model on Samsung Galaxy S4 with 5-inch screen. We then collected the smudges left behind on the device. He was rewarded with USD 50 after completing the entire task.

### 5.2 Camera and lighting

For smudge attacks to work best, camera and lighting conditions need to be carefully adjusted [5, 24]. In general, smudge attacks would work best when performed with a high-resolution camera and under a bright light. For taking a picture of the target device screen, we used Sony ILCE-7R camera with FE 35mm F2.8 ZA lens, and FOMEX D800p flash lighting system with 800W/s max flash output. All of those equipments cost about USD 3,200, and moderately sophisticated attackers (e.g., a government agency such as the FBI) should be able to afford them quite easily. Color pictures taken with that setting have a resolution of 7,360 × 4,912 pixels, with 24 bits of RGB color per pixel. The target device was placed in front of that camera attached to a paperboard (see Appendix C), and the angle between the camera lens and touchscreen was set to 30°. For this setup, we used a similar setting described in [24] with a small change in the distance between the camera and target device screen. Our camera and lighting setup is illustrated in Figure 19.

### 5.3 Threshold values for determining relevant segments

An important step in performing smug attack is the identification of segments (from detected smudge objects) that are part of an unlock pattern (see Section 3.3). From the detected smudge objects between two grid points, we need to determine whether that segment is part of a pattern. To do this, we count the number of smudge objects (i.e., detected edges) between given two grid points, and compare that number against a threshold value. For example, in Figure 4(b), the number of detected edges in the cyan box between

**Table 1: Comparison of the smug attack performance when different $n$-gram Markov model and smoothing technique combinations are used. "Avg. # of guessing attempts" shows the average when the number of guessing attempts was not limited; "Avg. # of guessing attempts (≤ 20)" shows the average considering only the successfully attacked cases (i.e., when patterns were cracked within 20 attempts)**

| | Add-2 | Add-3 | Add-4 | GT-2 | GT-3 |
|---|---|---|---|---|---|
| Avg. # of guessing attempts | 7,635.01 | 6,965.14 | 6,971.58 | **6,651.97** | 16,563.28 |
| Avg. # of guessing attempts (≤ 20) | 4.71 | 4.08 | **3.52** | 4.92 | 4.89 |
| Total # of cracked patterns (≤ 20) | 64 (68.82%) | 62 (66.67%) | 59 (63.44%) | **65 (69.89%)** | 64 (68.82%) |

points 5 and 6 is 44. If 44 is greater than a predefined threshold, we would accept this segment to be part of a given pattern. Otherwise, we would reject it. Therefore, it is important to choose an optimal threshold that maximizes the detection performance.

To that end, we suggest using the *ratio* of the "number of detected edges associated with a particular segment" to the "total number of detected edges in all pattern segments" as the threshold value. This is because the number of detected edges can vary significantly depending on the attack conditions such as camera settings or user pattern input behaviors.

We used a small volume of the training dataset to find an optimal threshold value for this binary classification test. Our results (see Appendix D) showed that 0.02 would be a reasonable threshold value to use in terms of *f-measure*. Although 0.04 is better than 0.02 in terms of *accuracy*, we selected 0.02 because in smug attacks the recall rate for edge detection is much more important than the precision rate; valid edges are integral in creating the list of possible patterns, including the correct pattern. Therefore, we used a more conservative threshold value to achieve better recall rate.

## 5.4 Markov model optimization based on smug attack performance

To improve the performance of the smug attack, it is essential to find an optimal $n$-gram Markov model. Therefore, we experimented with a number of reasonable $n$-gram Markov models with varying $n$ and smoothing techniques. As described in Section 3.4, probability of zero can be assigned to some $n$-grams since some patterns may not exist in the training set. This can be avoided by applying a well-known preprocessing technique called "smoothing," which assigns pseudo-counts to unseen events. It is a necessary process to avoid probability estimates that are zero for events that are not present in training samples. There are various smoothing techniques that can be used [14].

"Laplace smoothing" is a commonly used technique that works by adding $k$ pseudo-counts to the count of each event. In our experiments, we simply used a special case of Laplace smoothing with $k = 1$, which is popularly used in many applications such as language modeling. Notation Add-$n$ is used to refer to an $n$-gram Markov model used with one additional pseudo-count for an event. "Good-Turing smoothing" [11] is a more sophisticated technique, which estimates the total probability of events that appear $i$ times based on the total probability of events that appear exactly $i + 1$ times. In particular, the total probability for all unseen events in a given dataset can be estimated by the total probability of items that appear only once. We use GT-$n$ as the notation to refer to $n$-gram Markov model used with the Good-Turing smoothing technique.

To find the best performing Markov model configuration, we analyzed the performance of smug attack under various $n$-Markov models. First, we computed the average number of guessing attempts without limiting the number of guesses on failed attacks. Smug attack is successful if it unlocks a device within 20 guessing attempts; if not, it is considered as an unsuccessful attack. Second, we computed a more conservative average value, only considering the successfully attacked cases. Last, we counted the total number of successfully cracked patterns within 20 guessing attempts. Those results are presented in Table 1.

GT-2 showed the highest percentage of cracked patterns at 69.89% although the average number of guessing attempts for successfully cracked patterns (≤ 20) was 4.92 which is greater than the other models. Therefore, we performed all subsequent smug attacks using GT-2.

To measure the efficiency of smug attack, we analyzed the average time to complete each step in a smug attack. As described in Section 3, our smug attack implementation can roughly be divided into two phases: (1) image processing to infer possible patterns from smudges, and (2) sorting patterns based on the occurrence probabilities computed using an $n$-gram Markov-model. With our prototype implementation, the first step took about 8.31 seconds on average (with 0.56 standard deviation), and the second step took about 9.71 seconds on average (with 24.18 standard deviation). Hence, in total, it took only about 18.02 seconds on average to complete smug attack.

In the second step, the standard deviation is quite large because the processing time is highly dependent on the number of possible pattern candidates identified, and this number can vary quite a lot based on what the actual pattern is and the volume of smudges left behind.

# 6. SECOND RESULTS: SMUG ATTACK PERFORMANCE

This section presents the results collected from performing the fully optimized smug attack, GT-2 (see above), on all four different mobile usage scenarios, and shows how the smug attack performance can be affected by the increasing level of smudge obscurity.

## 6.1 Methodology

For the second experiment, we recruited seven male and five female (twelve in total) Android users to play the role of a victim. All participants were university students with varying hand sizes in their early and mid 20s.

In our study, we asked each participant to first unlock Samsung Galaxy S4 with 5 inch touchscreen (same as the first experiment) using a pattern that was randomly chosen from the testing pattern set of 93 patterns which are different from the 219 patterns used for constructing the $n$-gram Markov model, and perform three real-world mobile device usage scenarios (see Table 2): Calling someone, texting someone, and using the Facebook app. Those additional tasks mimic some common real-world mobile device usage scenarios. In the "using the Facebook app" task, for example, each participant was asked to freely use the Facebook app for a few minutes – this task was designed to closely resemble the participants' real-world use of their mobile devices. Each participant repeated this process 30 times with a different pattern. After each round, we

took a picture of the smudges left behind, cleaned the touchscreen, and reconfigured the device unlock setting with the next pattern. In consequence, 360 randomly selected patterns were tested among all 12 participants. Each participant was rewarded with USD 60 after completing all of those tasks, taking about 3 hours on average to complete everything. We never explained the research purposes to the participants.

**Table 2: Procedures for user tasks.**

| Task | Procedures |
|---|---|
| A. Unlocking screen only | 1. Draw the given pattern to unlock the device. |
| B. Calling | 1. Draw the given pattern to unlock the device.<br>2. Start the phone call app.<br>3. Enter a phone number randomly chosen from the address book by explicitly pressing all the numbers, and make a call. |
| C. Texting | 1. Draw the given pattern to unlock the device.<br>2. Start the instant messenger app.<br>3. Type given text messages randomly selected from real messages sent previously. |
| D. Using the Facebook app | 1. Draw the given pattern to unlock the device.<br>2. Start the Facebook app.<br>3. Perform a normal, daily action (e.g., writing a new post, replying to a comment, liking a post) on the Facebook app. |

## 6.2 Smug attack performance

To show how effective the optimized smug attack is, we compare the attack success rate of the smug attack against a pure Markov model-based guessing attack. First, we evaluated the performance of several Markov models, without any smudge support, on the first naive scenario where the participants merely unlocked phones. The 3-gram Markov model with Laplace smoothing (see the top 20 most likely used patterns in Appendix F) produced the best results, cracking 48 out of 360 patterns (13.33%) with 7.31 guesses on average ($\leq 20$). This result alone is significant, but the smug attack performance was even more devastatingly significant, cracking 267 patterns (74.17%) from the same pattern set ($p < 0.001$, FET).

20 of the patterns cracked by the pure Markov model-based guessing attack had a length of 4 segments, 17 had a length of 5 segments, and 11 had a length of 7 segments. In contrast, 107 of the patterns cracked by the smug attack had a length of 4 segments, 48 had a length of 5 segments, 42 had a length of 6 segments, 25 had a length of 7 segments, 24 had a length of 8 segments, and 21 had a length of 9 segments. This shows that the pure Markov model-based guessing attack is relatively less effective against longer patterns (length > 5 segments).

## 6.3 Effects of smudge obscurity

This section analyzes the effects of different mobile device usage scenarios on the performance of the smug attack. Progressing from the naive device unlocking task to the Facebook task (see Table 2), the number of actions the participant had to perform increased, and as a result, the volume of obscuring smudges left on the touchscreen increased. Intuitively, the performance of smug attack should downgrade as the volume of obscuring smudges increases, and this is exactly what we observed from those different real-world usage scenarios. The same smug attack, with GT-2 configuration, was performed on the smudges collected from all of the four tasks; however, we did not use the *longest pattern first trial* strategy on the calling, texting and Facebook tasks because additional smudge objects that are not relevant to the user's original pattern could have been created while performing those tasks (see Table 2).

The smug attack success rates for unlocking device, calling, texting, and Facebook tasks were 74.17% (the average percentage among 12 participants were $\sigma$ = 10.90%), 52.50% ($\sigma$ = 11.72%), 37.22% ($\sigma$ = 9.89%), and 31.94% ($\sigma$ = 9.95%), respectively, showing a gradual decrease in the smug attack performance with the increase in the volume of obscuring smudges (see Table 3). To measure the statistical significance between those attack success rates, we performed the Fisher's Exact Test (FET) with Bonferroni correction. The attack success rate for the naive device unlocking task was significantly higher than the rates for all other three tasks (all $p < 0.001$, pairwise corrected FET). Similarly, the success rate for the calling task showed statistically significant differences against both the texting task and the Facebook task ($p < 0.005$, pairwise corrected FET). The average numbers of guessing attempts were 3.79, 4.43, 5.36, and 4.82 for the four tasks, respectively. Considering that there is about 42.23% difference in the attack success rate between the naive device unlocking task and the Facebook task, it is clear that obscuring smudges have a significant impact on the smug attack performance or on the performance of smudge-supported attacks in general.

## 6.4 False positive and false negative analysis

For more detailed investigation on the effects of smudge obscurity, we analyzed the characteristics of frequently misclassified segments in each of the three (calling, texting, and Facebook) tasks.

Across all the three additional tasks, we computed the false positive and false negative rates for each pattern segment, and compared their rates with the rates computed for the naive device unlocking task. We performed the Fisher's Exact Test to identify rate differences between pattern segments that are statistically significant (see Appendix E). "FP segment" denotes a non-existing segment that is identified more frequently as an existing segment in one of the three additional tasks compared to the device unlocking task. "FN segment" denotes an existing segment that is identified more frequently as a non-existing segment in one of the three additional tasks compared to the device unlocking task.

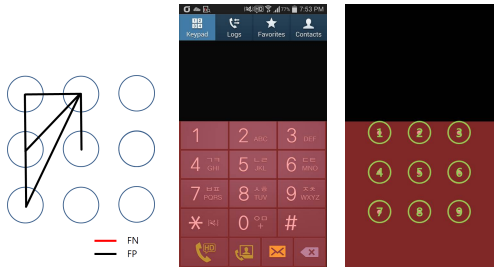
For the calling task, we found just 6 FP segments, which were mainly located in the upper left hand side of the pattern grid (see Figure 9). It is hard to explain why such non-existing segments were more frequently identified as existing segments when the smudge obscurity increased. It might be due to the distribution of digits in the phone numbers that the participant used to make calls (we did ask participants to call a different person each time).

For the texting task, we found 4 FN segments and 9 FP segments, which were mainly located around the lower part of the pattern grid (see Figure 10). This is because for texting the participant mainly interacts with the on-screen keyboard, which is located near the lower part of the screen, affecting the lower part of the smudges that were left from drawing a pattern.

For the Facebook task, we found 3 FN segments and 8 FP segments, which were mostly located on the right hand side of the pattern grid (see Figure 11). We believe this is due to the participant mainly scrolling up and down to view posts on his timeline, which involves (in most cases) using the right thumb and flicking up or down on the right hand side of the touchscreen.

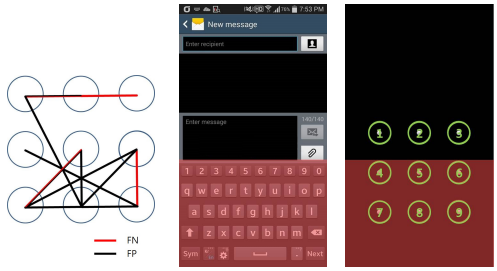
Table 3: Comparison of the smug attack performance across the four device usage scenarios.

| | Unlocking only | Calling | Texting | Facebook |
|---|---|---|---|---|
| Avg. # of guessing attempts | 4,634.66 | 6,811.83 | 9,783.01 | 13,130.74 |
| Avg. # of guessing attempts (≤ 20) | 3.79 | 4.43 | 5.36 | 4.82 |
| Total # of cracked patterns (≤ 20) | 267 (74.17%) | 189 (52.50%) | 134 (37.22%) | 115 (31.94%) |



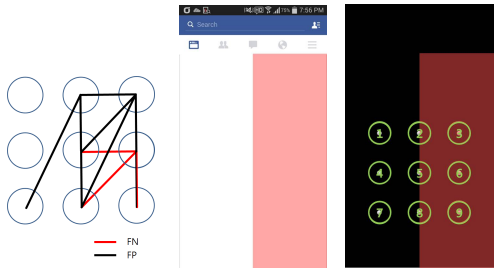

(a) Misclassified (b) Task area (c) Overlapped

**Figure 9: Frequently misclassified segments found for the "calling" task.**




(a) Misclassified (b) Task area (c) Overlapped

**Figure 10: Frequently misclassified segments found for the "messaging" task.**




(a) Misclassified (b) Task area (c) Overlapped

**Figure 11: Frequently misclassified segments found for the "Using the Facebook app" task.**

Those results suggest a clear limitation of smudge-based inference attacks, which will not perform well if a touchscreen has too many obscuring smudges left behind (e.g., a device that has been used by the victim for an hour or more without being cleaned).

Moreover, those results could be used to optimize the tool by adjusting the smug attack parameters based on the tasks that a victim has performed on the stolen mobile device. For example, in the segment decision procedure (see Section 3.3), a high threshold value can be used for FP segments, and a low threshold value can be used for FN segments. Further work is needed to try this kind of optimization technique on those parameters.

# 7. MITIGATION STRATEGIES

In this section, we discuss three possible mitigation techniques for smug attack. From the three, we implemented the first technique that deliberately adds obscuring smudges by mandating users to draw an additional random pattern upon log in, and evaluated its effectiveness against smug attack. We explain this technique first.

## 7.1 Adding obscurity

Our second experiment results (see Table 3) showed that adding smudge obscurity by asking the participant to perform different tasks on a device can significantly degrade smug attack performance. For instance, the performance decreased from 74.17% of cracked patterns in the naive usage scenario to 31.94% in the Facebook app usage scenario where participants left more irrelevant smudges on the touchscreen. Such obscuring techniques can be used to our advantage in mitigating smug attack: we could, for example, ask users to draw additional random segments, which would leave more redundant smudges on the touchscreen to obscure the visibility. This technique is visualized in Figure 12. A user is first asked to draw the actual unlock pattern and this is displayed in green. After unlocking his or her device, the user is then asked to draw a given random pattern, which is shown in white. This second pattern is a random pattern that is not stored anywhere—its purpose is to simply get the user to draw more segments and leave more smudges on the screen.

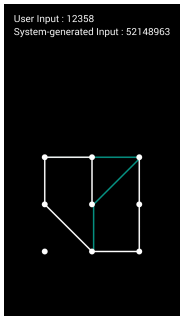



**Figure 12: An example pattern with an additional random pattern drawn on top of it.**

We implemented this obscurity based mitigation technique, and asked the same 12 participants from the second experiment (see Section 6) to draw each of the 30 patterns as well as a given random (obscuring) pattern. For this experiment, the participants were merely asked to perform the first naive screen unlock task. Hence, we compared the new smug attack performance against the baseline performance, GT-2 (13.33% of cracked patterns), shown in Table 3. The smug attack was performed without the heuristic that tries longer patterns first (see Section 3), expecting that such a heuristic could be less effective when there are obscuring segments.

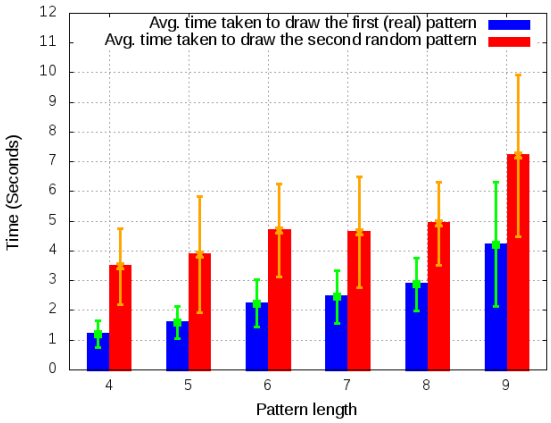



**Figure 13: The average time taken to draw the real pattern (in blue), and the average time taken to draw the random pattern (in red) for pattern length between 4 and 9.**

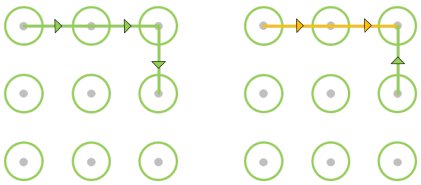
With the obscuring technique in place, the total number of cracked patterns was 124 (34.44%), which is close to the smug attack success rate (31.94%) for the Facebook scenario. This is a huge improvement from the original result (without obscuring technique), where 267 (74.17%) patterns were cracked ($p < 0.001$, FET). The average number of guessing attempts made for patterns that were cracked increased significantly from 3.79 (without obscuring technique) to 5.24. Those results clearly demonstrate the effectiveness of the obscuring technique.

The key advantage of the obscuring technique is that it ensures backward compatibility such that existing patterns can be used without making any change unlike the existing smudge attack mitigation schemes (e.g., [17]). The only additional logic introduced is the generation of a random pattern at run time, and requiring users to draw a random pattern after unlocking their device. The usability compromise here is the additional pattern that user has to draw after unlocking their device. The graph in Figure 13 shows the average time taken for the participant to draw the real pattern (blue), and the additional time taken to draw a given random pattern. On average, it took the participant about 2-3 times longer to draw a random pattern for each pattern length (4-9), which is the login time tradeoff for increasing obscurity (adding security). For random patterns with length between 4 and 8, the participant took about 3-5 seconds on average to draw them, which would be a reasonable delay. However, for random patterns of length 9, the participants took about 7 seconds on average to draw them—this delay might annoy users. Hence, random patterns with length equal to 9 are not ideal candidates for this mitigation technique.

## 7.2 Allowing repeated points

The performance of smug attack heavily depends on the size of the possible pattern space: i.e., the larger the pattern space, the more challenging it is for smug attack to make accurate guesses. Hence, an intuitive strategy is to increase the possible pattern space.

Android enforces a pattern creation policy that prohibits points and lines from being used more than once in a pattern. In Figure 14(a), for example, a possible pattern that contains points 1, 2, 3 and 6 can be inferred from smudge residuals as either "1236" or "6321". Smug attack only needs to try two different possibilities to guess that pattern, which is straightforward. However, if we amend the Android pattern creation policy to allow multiple use of points and segments in a given pattern, we can increase the workload of smug attack significantly by increasing the possible pattern space. We can see a pattern in Figure 14(b) that appears to contain the same points (1, 2, 3 and 6) inferred from visible smudge residuals, but the actual pattern is "632123", which is not only longer but much more difficult to infer from analyzing smudges. With this amended policy, smug attack will now have to consider all possible patterns that reuse points and segments, e.g., "1236321", "123632", "21236", and so on. With the current policy, there are 389,112 possible patterns, but with the amended policy, we can significantly increase the pattern space to 1,826,902,512. This is about 4,695 times larger than the original pattern space. Considering that Android limits the number of failed attempts allowed to 20, this huge increase in the pattern space will make it difficult for smug attack to succeed.



(a) 1-2-3-6 (b) 6-3-2-1-2-3

**Figure 14: Non-repeated vs repeated points**

## 7.3 Changing the pattern grid location

Image processing techniques used by smug attack rely on the location of the $3 \times 3$ grid being static for a given Android device. It looks for smudges that match the location of the points and uses them to identify possible patterns. Our smug attack tool uses the pre-stored template images (see Section 3) to find the exact grid location from the input picture of the touchscreen, and to identify smudges that match the location of the points and possible line segments. Our third mitigation strategy exploits this operation and suggests altering the location of the grid slightly every time a user tries to unlock a device. An example is shown in Figure 15. When a target device is wrongly positioned, the pattern area could be mismatched by our smug attack tool.

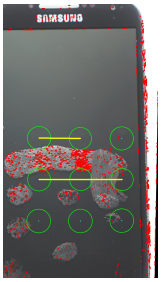



**Figure 15: An example of mismatched pattern area.**

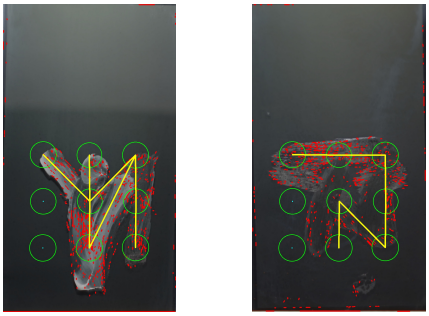
Based on our experience in developing smudge attacks, it will be hard for the smug attack tool to extract patterns from smudges as they will no longer match the points from a given template such as the original grid. In fact, Schneegass et al. [17] have proposed a similar mitigation technique that uses geometric image transformations such as grid translation, rotation, or scaling. Such techniques can be effective in mitigating template-based smudge attacks. However, it is likely to downgrade the usability of pattern drawing process as this technique is quite similar to random keyboard arrangements used to defeat keyloggers.

# 8. DISCUSSION

This section discusses the smug attack performance and its limitations.

## 8.1 Effectiveness of smug attacks

To show the effectiveness of smug attack, we compared its performance against the pure Markov model-based guessing attack (see Section 6.2). For the naive device unlocking scenario, the fully optimized smug attack (GT-2) significantly outperformed the pure guessing attack in the attack success rate, successfully cracking 267 (out of 360) more patterns. The difference in the attack success rates was about 60.84% ($p < 0.001$, FET). We also demonstrated that the pure Markov model is not so effective against patterns longer than length 5 whereas smug attack is much more capable of cracking longer patterns. Moreover, we have shown that our tool can effectively recognize patterns that are hard to see with naked human eyes (see Figure 16). Smudge attacks can significantly boost the performance of a probabilistic password model (e.g., the $n$-gram Markov model), and can be used to develop a fully automated guessing tool. Even when obscuring smudges were added under the Facebook scenario, the proposed smug attack, at 31.94%, still outperformed the pure guessing attack, at 13.33%, in the attack success rate.



(a) 15369 (b) 123695

**Figure 16: Examples of patterns that cannot be easily recognized by the naked human eyes.**

## 8.2 Limitations of smug attacks

Despite the performance boost, Table 3 also shows a clear limitation of smug attacks where the attack success rate significantly decreased as the tasks became more complex, requiring the participant to perform more actions. The attack success rate (patterns cracked within 20 guessing attempts) started from 74.17% when the task was to merely unlock the given device, and that rate decreased to 52.50%, 37.22%, and 31.94% as the participant was asked to also make a call, send text messages, and use the Facebook app, respectively. This reduction in the effectiveness of smug attack is due to the increased volume of obscuring smudges, and more relevant smudges being erased.

Our real-world dataset- and implementation-based findings contrast with the speculative findings from previous literature that only highlighted the strong effectiveness of smudge attacks based on user feedback. Our results, for the first time, demonstrate how obscurity can affect the performance of smudge attacks based on real data analyses.

Moreover, our results showed that physical characteristics and/or pattern drawing behaviors of individuals could impact smug attack success rates, creating variances in the results. With the calling task ($\mu = 52.50\%$, $\sigma = 11.72\%$), in particular, we observed high variances in the results (even though each participant had to draw different pattern sets). As part of the future work, we will study how personalization of smug attack configurations could affect its performance.

## 8.3 Mitigation through adding obscurity

In Section 7, we discussed three possible mitigation techniques for smug attack. From those three techniques, we implemented and evaluated the obscurity based mitigation technique where users are also asked to draw a random pattern upon log in to deliberately add obscuring smudges. Our experiment results showed that this obscuring technique is highly effective (confirming our observations on the limitations of smug attack), but the main tradeoff in usability is the time taken for a user to draw the second random pattern, which takes about 3-5 seconds on average. We could improve user experience by selectively asking users to enter the second random pattern, e.g., only when a user is at a public place like libraries or cafes. Location technologies like GPS can be used to automatically detect when a user is at a public place, and enable it. Users do not have to remember anything extra.

# 9. RELATED WORK

In this section, we summarize recent research findings on attacks performed against popularly used authentication mechanisms on mobile devices: (1) smudge attacks and (2) guessing attacks.

Smudge attacks guess a user's password or pattern using fingerprint traces left on the touchscreen of a target mobile device. Aviv et al. [5] discussed the feasibility of performing smudge attacks to guess Android patterns, and experimented with different camera orientations and light positions. Their attack method, however, was not fully automated, and their results were based on the participants' self reports on the perceived difficulty of identifying patterns from smudges visible on a touchscreen.

Several researchers have worked on defense mechanism for smudge attacks. Zezschwitz et al. [24] proposed three new pattern grid layouts, and evaluated their usability and security through a lab experiment. Kwon et al. [12] suggested the use of a small grid layout with mandating user interactions to remove smudge traces. Schneegass et al. [17] proposed a geometrically transformed graphical password scheme for a similar purpose. Their security evaluation, however, was conducted using 32 synthetically-generated graphical passwords in a lab environment. None of those research groups developed a fully automated tool for performing smudge attacks or guessing attacks against graphical passwords.

Guessing attack is one of the most commonly performed attacks on password-based authentication schemes. The main goal of this attack is to build a comprehensive dictionary for cracking passwords efficiently. Since the distribution of user chosen passwords (including Android patterns) tends to be heavily skewed toward small number of popularly used passwords, they are generally vulnerable to guessing attacks. For example, Van Oorschot et al. [16] showed that the actual password space of "Draw-A-Secret" graphical passwords tends to be significantly smaller than the theoretically full password space. For Android patterns, Uellenbeck et al. [21] particularly conducted a survey to collect user patterns and found that their actual pattern space is much smaller than the theoretical space. Andriotis et al. [3] also analyzed the Android pattern security based on user chosen patterns. They conducted an online survey to collect user patterns, asking participants to create patterns that are easy-to-remember and secure. Their results showed that user chosen patterns are biased; for example, memorable pattern shapes such as "⌞" or "¬" were popularly used, and users frequently chose the upper-left grid point as the starting point

in their patterns. Song et al. [19] collected a small number of real user patterns, and constructed an $n$-gram Markov model with the collected data. Based on the Markov model, they presented a list of most likely occurring real-world patterns and suggested that this list could be used as a dictionary for guessing patterns. Intuitively, it is believed that the use of password meter [19] and bigger grid layout [4] could be helpful to improve the security of user chosen patterns but existing studies [4, 19] demonstrated that their impacts are practically limited. For example, even with the 4×4 grid, 19% of patterns can successfully be cracked, which is comparable to 15% of the attack success rate with the 3×3 grid [4].

Aviv et al. [5] previously claimed that smudge data could be combined with statistical information about human behaviors such as pattern usage distribution to perform an effective attack. This paper finally implements this idea, and demonstrates the effectiveness of the combined attack based on a real-world pattern dataset.

# 10. CONCLUSION

This paper studies the effectiveness of combining guessing attacks with smudge attacks on unlocking Android devices within 20 guessing attempts (this is the number of consecutive fail attempts allowed on Android). We trained a Markov model-based guessing attack using 219 (70%) of 312 real-world patterns, and recruited 12 participants to individually draw 30 patterns which were randomly chosen from the remaining 30% of those patterns on Samsung Galaxy S4 in a lab environment.

Our results showed that smudge attacks can indeed boost the performance of guessing attacks by providing a way to pre-compute only the possible pattern candidates based on the detected segments. In the case of a naive device unlock scenario, the attack performance significantly improved from 13.33% when the pure guessing attack was performed alone to 74.17% when the smug attack was performed. Even when obscuring smudges were added under a more complex scenario that involved the use of the Facebook app, our smug attack, at 31.94%, still outperformed the pure guessing attack. However, the limitation of smudge-based attacks was also clear, showing that obscuring smudges can significantly downgrade their performance, and mitigation techniques should be designed to help users add obscurity.

The proposed technique, with some parameter adjustments, could be used to effectively crack other types of authentication mechanisms used on touchscreens (e.g., a PIN or password). As part of future work, we plan to further investigate the performance of the combined attack on PINs, experimenting with 4- and 6-digit PINs.

# Acknowledgement

This work was supported in part by the ITRC (IITP-2016-R0992-16-1006), the MSIP/IITP (R-20160222-002755) and the MISP (R2215-16-1005). Note that Hyoungshick Kim is the corresponding author.

# 11. REFERENCES

[1] OpenCV. http://docs.opencv.org/, 2015.

[2] ANDRIOTIS, P., TRYFONAS, T., AND OIKONOMOU, G. Complexity metrics and user strength perceptions of the pattern-lock graphical authentication method. In *Human Aspects of Information Security, Privacy, and Trust* (2014), Springer, pp. 115–126.

[3] ANDRIOTIS, P., TRYFONAS, T., OIKONOMOU, G., AND YILDIZ, C. A pilot study on the security of pattern screen-lock methods and soft side channel attacks. In *Proceedings of the 6th ACM conference on Security and Privacy in Wireless and Mobile Networks* (2013).

[4] AVIV, A. J., BUDZITOWSKI, D., AND KUBER, R. Is bigger better? comparing user-generated passwords on 3x3 vs. 4x4 grid sizes for android's pattern unlock. In *Proceedings of the 31st Annual Computer Security Applications Conference* (2015).

[5] AVIV, A. J., GIBSON, K., MOSSOP, E., BLAZE, M., AND SMITH, J. M. Smudge Attacks on Smartphone Touch Screens. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (2010).

[6] AVIV, A. J., SAPP, B., BLAZE, M., AND SMITH, J. M. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012).

[7] BALLARD, D. H. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition 13*, 2 (1981), 111–122.

[8] BBC NEWS. '60,000' devices are left in cabs. Online. Access at: http://news.bbc.co.uk/2/hi/technology/7620569.stm, 2008.

[9] BBC NEWS. FBI-Apple case: Investigators break into dead San Bernardino gunman's iPhone. Online. Access at: http://www.bbc.com/news/world-us-canada-35914195, 2016.

[10] CANNY, J. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6 (1986), 679–698.

[11] GALE, W. A. Good-turing smoothing without tears. *Journal of Quantitative Linguistics* (1995).

[12] KWON, T., AND NA, S. Tinylock: Affordable defense against smudge attacks on smartphone pattern lock systems. *Computers & Security 42* (2014), 137–150.

[13] LEE, J., HARALICK, R., AND SHAPIRO, L. Morphologic edge detection. *IEEE Journal of Robotics and Automation 3*, 2 (1987), 142–156.

[14] MA, J., YANG, W., LUO, M., AND LI, N. A study of probabilistic password models. In *IEEE Symposium on Security and Privacy* (2014).

[15] MATAS, J., GALAMBOS, C., AND KITTLER, J. Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding 78*, 1 (2000), 119–137.

[16] OORSCHOT, P. C. V., AND THORPE, J. On predictive models and user-drawn graphical passwords. *ACM Transactions on Information and System Security 10*, 4 (2008), 5:1–5:33.

[17] SCHNEEGASS, S., STEIMLE, F., BULLING, A., ALT, F., AND SCHMIDT, A. Smudgesafe: Geometric image transformations for smudge-resistant user authentication. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing* (2014).

[18] SHANNON, C. E. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review 5*, 1 (2001), 3–55.

[19] SONG, Y., CHO, G., OH, S., KIM, H., AND HUH, J. H. On the Effectiveness of Pattern Lock Strength Meters: Measuring the Strength of Real World Pattern Locks. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (2015).

[20] TAO, H., AND ADAMS, C. Pass-go: A proposal to improve the usability of graphical passwords. *International Journal of Network Security 7*, 2 (2008), 273–292.

[21] UELLENBECK, S., DÜRMUTH, M., WOLF, C., AND HOLZ, T. Quantifying the security of graphical passwords: the case of android unlock patterns. In *Proceedings of the 20th ACM conference on Computer and Communications Security* (2013).

[22] VAN BRUGGEN, D., LIU, S., KAJZER, M., STRIEGEL, A., CROWELL, C. R., AND D'ARCY, J. Modifying Smartphone User Locking Behavior. In *Proceedings of the Ninth Symposium on Usable Privacy and Security* (2013).

[23] VON ZEZSCHWITZ, E., DUNPHY, P., AND DE LUCA, A. Patterns in the Wild: A Field Study of the Usability of Pattern and Pin-based Authentication on Mobile Devices. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services* (2013).

[24] VON ZEZSCHWITZ, E., KOSLOW, A., DE LUCA, A., AND HUSSMANN, H. Making graphic-based authentication secure against smudge attacks. In *Proceedings of the International Conference on Intelligent User Interfaces* (2013).

[25] ZAKARIA, N. H., GRIFFITHS, D., BROSTOFF, S., AND YAN, J. Shoulder Surfing Defence for Recall-based Graphical Passwords. In *Proceedings of the Seventh Symposium on Usable Privacy and Security* (2011).

# APPENDIX

## A. PATTERN LOCK AUTHENTICATION IN ANDROID

Figure 17 shows a typical interface of pattern lock authentication in Android.

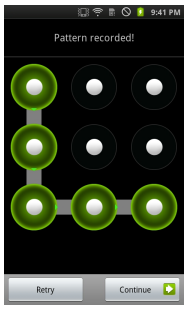
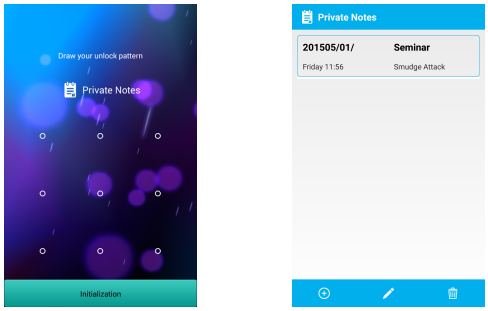



**Figure 17: Pattern lock authentication in Android.**

## B. ANDROID APP FOR DATA COLLECTION

To achieve complete ecological validity for real-world patterns, we developed an independent application called Private Notes (see Figure 18), which allows users to encrypt their personal notes and made it available on Google Play (https://play.google.com/store/apps/details?id=com.Seclab.Notes).




(a) Pattern setup (b) User interface

**Figure 18: Screenshots of Private Notes.**

## C. EXPERIMENTAL SETUP FOR CAMERA AND LIGHTNING

For smudge attacks to work best, we carefully tested several camera and lighting conditions. The target device was placed in front of that camera attached to a paperboard (see Figure 19), and the angle between the camera lens and touchscreen was set to 30°.

## D. EXPERIMENT RESULTS FOR DETERMINING THRESHOLD VALUES IN THE SEGMENT DECISION PROCEDURE

We associated "existing segments" with positive answers ($P$) and "non-existing segments" with negative answers ($N$). The definition of true positive ($TP$), false positive ($FP$), true negative ($TN$), and false negative ($FN$) can be summarized as follows:

- $TP$: "existing segments" correctly classified as "existing segments";

**Figure 19: To take a picture needed for smug attack, a smartphone was placed under a bright light source, and in front of a high-resolution camera.**

- $FP$: "non-existing segments" incorrectly classified as "existing segments";
- $TN$: "non-existing segments" correctly classified as "non-existing segments";
- $FN$: "existing segments" incorrectly classified as "non-existing segments".

To evaluate the performance of the segment decision procedure with varying threshold values, we used the following four measures:

- **Accuracy**: the proportion of correctly classified segments; $(TP + TN)/(P + N)$
- **Precision**: the proportion of segments classified as "existing segments" that actually are "existing segments"; $(TP)/(TP + FP)$
- **Recall**: the proportion of "existing segments" that were accurately classified; $(TP)/(TP + FN)$
- **F-measure**: the harmonic mean of *precision* and *recall*; $(2 * Precision * Recall)/(Precision + Recall)$

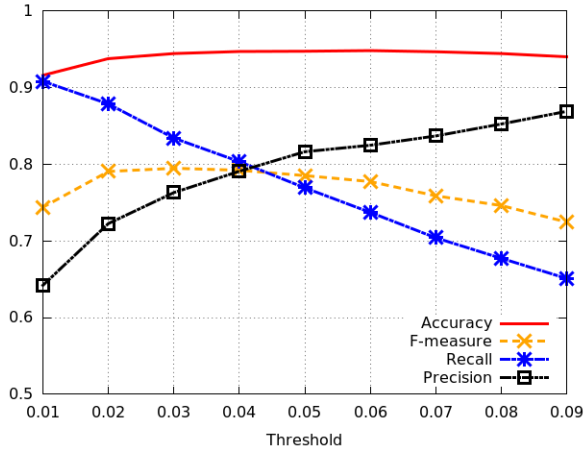



**Figure 20: Test performance with varying threshold values.**

## E. FISHER'S EXACT TEST RESULTS FOR FALSE POSITIVE AND FALSE NEGATIVE ANALYSIS

We performed the Fisher's Exact Test to identify rate differences between pattern segments that are statistically significant (see Table 4). Table 4 shows the test results where significantly different cases are represented in bold font. "FP segment" denotes a non-existing segment that is identified more frequently as an existing segment in one of the three additional tasks compared to the device unlocking task. "FN segment" denotes an existing segment that is identified more frequently as a non-existing segment in one of the three additional tasks compared to the device unlocking task.

**Table 4: Fisher's Exact Test results for the comparison of the false positive and negative rates between unlocking screen and each of post tasks (calling, messaging or using Facebook). If p-value $\leq 0.05$, we use a bold font to represent its significance.**

| Segment | Call | | Message | | Facebook | |
|---|---|---|---|---|---|---|
| | FN | FP | FN | FP | FN | FP |
| (1, 2) | **0.01** | 1.00 | **0.00** | **0.01** | 0.83 | 1.00 |
| (1, 4) | **0.00** | 0.48 | 1.00 | 0.48 | 0.58 | 1.00 |
| (1, 5) | 0.14 | 1.00 | 1.00 | 1.00 | 0.21 | 1.00 |
| (1, 6) | 0.18 | 1.00 | 0.23 | 1.00 | 1.00 | 1.00 |
| (1, 8) | 0.07 | 1.00 | **0.03** | 1.00 | 0.45 | 1.00 |
| (2, 3) | 0.16 | 1.00 | 0.09 | **0.00** | **0.00** | 0.05 |
| (2, 4) | **0.03** | 1.00 | 0.79 | 0.66 | 0.36 | 0.59 |
| (2, 5) | **0.02** | 1.00 | 0.05 | 1.00 | **0.00** | 1.00 |
| (2, 6) | 1.00 | 1.00 | 1.00 | 1.00 | 0.49 | 1.00 |
| (2, 7) | **0.02** | 1.00 | 0.09 | 1.00 | **0.00** | 1.00 |
| (2, 9) | 0.43 | 1.00 | 0.65 | 1.00 | 0.80 | 1.00 |
| (3, 4) | 0.36 | 1.00 | 0.68 | 1.00 | 0.15 | 1.00 |
| (3, 5) | 0.79 | 1.00 | 0.79 | 1.00 | **0.00** | 1.00 |
| (3, 6) | 1.00 | 0.41 | 1.00 | 0.24 | **0.00** | 1.00 |
| (3, 8) | 0.56 | 1.00 | 0.07 | 1.00 | **0.00** | 1.00 |
| (4, 5) | 0.70 | 1.00 | 0.05 | 0.65 | 0.45 | 1.00 |
| (4, 7) | **0.00** | 0.48 | 0.09 | 0.13 | 0.15 | 0.41 |
| (4, 8) | 0.08 | 0.48 | 0.13 | 1.00 | 0.32 | 0.48 |
| (4, 9) | 0.56 | 1.00 | **0.01** | 1.00 | 0.28 | 1.00 |
| (5, 6) | 1.00 | 1.00 | 1.00 | 0.35 | 0.66 | **0.00** |
| (5, 7) | 1.00 | 1.00 | **0.00** | **0.04** | 1.00 | 0.48 |
| (5, 8) | 0.05 | 1.00 | **0.01** | 0.24 | **0.00** | 0.66 |
| (5, 9) | 0.62 | 1.00 | 0.40 | 0.18 | 0.82 | 0.18 |
| (6, 7) | 1.00 | 1.00 | **0.00** | 1.00 | 0.21 | 1.00 |
| (6, 8) | 1.00 | 0.60 | **0.00** | 1.00 | 1.00 | **0.00** |
| (6, 9) | 1.00 | 0.24 | 0.15 | **0.00** | **0.00** | 0.53 |
| (7, 8) | 1.00 | 1.00 | **0.00** | 0.23 | 1.00 | 0.29 |
| (8, 9) | 1.00 | 1.00 | **0.00** | 0.23 | 0.23 | **0.00** |

## F. TOP 20 MOST LIKELY USED PATTERNS

In this section, we look at the characteristics of popularly used real-world patterns based on the 312 real-world patterns collected through Private Notes. Instead of using GT-2 (which is what we used in the optimized smug attack), we used the 3-gram Markov model with the Laplace smoothing technique to compute occurrence probabilities of all possible patterns, which overall produced the best results in the case of pure Markov model-based guessing attack. The top 20 most popularly used Android lock patterns are shown in Figure 21, and here are some interesting observations:

**Short length.** Except for the 3rd, 17th and 18th patterns, the lengths of all other patterns are less than or equal to 5.

**Small number of turns.** Except for the 15th, 18th and 19th patterns, all other patterns have just one or two turns in them.

**No crossing points.** All of those patterns have no crossing points.

**Popular directions.** Except for the 20th pattern, the rest of all patterns start from the left side of the grid and move to the right side. Also, all of them start from the top of the grid and move to the bottom. We believe that those characteristics are strongly related to the directions in which many written languages are interpreted—we looked at the country information of the users of Private Notes available on Google Play to confirm this. Such trends indicate that information about users' geographical location and language may provide useful guessing hints for adversaries.

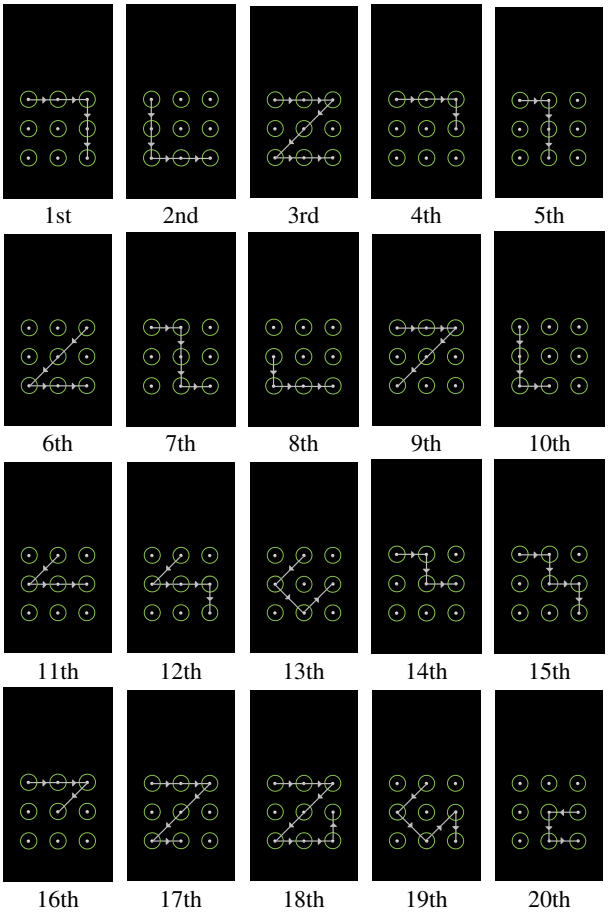



**Figure 21: The top 20 most likely used patterns identified using the Markov model.**

Based on those characteristics, we surmise that many users prefer using simple patterns with small number of turns that can be drawn quickly. Such user patterns would be susceptible to smudge attacks, shoulder-surfing attacks, and guessing attacks.