Additional Material

Adam J. Aviv
Ph.D. Candidate
University of Pennsylvania

- Adam J. Aviv, Katherine Gibson, Evan Mossop, Matthew Blaze, and Jonathan M. Smith. Smudge Attacks on Smartphone Touchscreens. In *4th Usenix Workshop on Offensive Technologies (WOOT)*, 2010.

- Adam J. Aviv, Matt Blaze, Jonathan M. Smith, and Micah Sherr. Privacy-Aware Message Exchanges for Geographically Routed Human Movement Networks . Under Submission, Submitted November 2012.

- Adam Aviv, Vin Mannino, Thanat Olwlar, Seth Shannin, and Kevin Xu. *PennOS: UNIX-like Operating System Simulation.* CIS-380 Operating Systems Course Project.

Smudge Attacks on Smartphone Touch Screens

Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith
Department of Computer and Information Science – University of Pennsylvania
{aviv,gibsonk,emossop,blaze,jms}@cis.upenn.edu

Abstract

Touch screens are an increasingly common feature on personal computing devices, especially smartphones, where size and user interface advantages accrue from consolidating multiple hardware components (keyboard, number pad, *etc.*) into a single software definable user interface. Oily residues, or *smudges*, on the touch screen surface, are one side effect of touches from which frequently used patterns such as a graphical password might be inferred.

In this paper we examine the feasibility of such *smudge attacks* on touch screens for smartphones, and focus our analysis on the Android password pattern. We first investigate the conditions (*e.g.*, lighting and camera orientation) under which smudges are easily extracted. In the vast majority of settings, partial or complete patterns are easily retrieved. We also emulate usage situations that interfere with pattern identification, and show that pattern smudges continue to be recognizable. Finally, we provide a preliminary analysis of applying the information learned in a smudge attack to guessing an Android password pattern.

1 Introduction

Personal computing devices now commonly use touch screen inputs with application-defined interactions that provide a more intuitive experience than hardware keyboards or number pads. Touch screens are touched, so oily residues, or *smudges*, remain on the screen as a side effect. Latent smudges may be usable to infer recently and frequently touched areas of the screen – a form of information leakage.

This paper explores the feasibility of *smudge attacks*, where an attacker, by inspection of smudges, attempts to extract sensitive information about recent user input. We provide initial analysis of the capabilities of an attacker who wishes to execute a smudge attack. While this analysis is restricted to smartphone touch screens, specifically attacks against the Android password pattern, smudge attacks may apply to a significantly larger set of devices, ranging from touch screen ATMs and DRE voting machines to touch screen PIN entry systems in convenience stores.

We believe smudge attacks are a threat for three reasons. First, smudges are surprisingly[1] persistent in time. Second, it is surprisingly difficult to incidentally obscure or delete smudges through wiping or pocketing the device. Third and finally, collecting and analyzing oily residue smudges can be done with readily-available equipment such as a camera and a computer[2].

To explore the feasibility of smudge attacks against the Android password pattern, our analysis begins by evaluating the conditions by which smudges can be photographically extracted from smartphone touch screen surfaces. We consider a variety of lighting angles and light sources as well as various camera angles with respect to the orientation of the phone. Our results are extremely encouraging: in one experiment, the pattern was partially identifiable in 92% and fully in 68% of the tested lighting and camera setups. Even in our worst performing experiment, under less than ideal pattern entry conditions, the pattern can be partially extracted in 37% of the setups and fully in 14% of them.

We also consider simulated user usage scenarios based on expected applications, such as making a phone call, and if the pattern entry occurred prior to or post application usage. Even still, partial or complete patterns are easily extracted. We also consider incidental contact with clothing, such as the phone being placed in a pocket; information about the pattern can still be retrieved. Finally, we provide preliminary analysis of applying a smudge attack to the Android password pattern and how the information learned can be used to guess likely passwords.

Next, in Sec. 2, we provide our threat model, followed by background on the Android password pattern in Sec. 3. Our experimental setup is presented in Sec. 4, including a primer on lighting and photography. Experimental results are presented in Sec. 5, and a discussion of applying a smudge attack to the Android pattern password is presented in Sec. 6. Related work is provided in Sec. 7, and we conclude in Sec. 8.

[1] One smartphone in our study retained a smudge for longer than a month without any significant deterioration in an attacker's collection capabilities.

[2] We used a commercial photo editing package to adjust lighting and color contrast, only, but software included with most operating systems is more than sufficient for this purpose.

2 Threat Model

We consider two styles of attacker, passive and active. A *passive attacker* operates at a distance, while an *active attacker* has physical control of the device.

A passive attacker who wishes to collect smartphone touch screen smudges may control the camera angle, given the attacker controls the camera setup, but the smartphone is in possession of its user. The attacker has no control of the places the user takes the smartphone, and thus cannot control lighting conditions or the angle of the phone with respect to the camera. The attacker can only hope for an opportunity to arise where the conditions are right for good collection. An active attacker, however, is capable of controlling the lighting conditions and is allowed to alter the touch screen to increase retrieval rate. This could include, for example, cleaning the screen prior to the user input, or simply moving the touch screen to be at a particular angle with respect to the camera.

For the purposes of our experiment, we make a strong assumption about the attacker's "activeness;" she is in possession of the device, either surreptitiously or by confiscation, and is capable of fully controlling the lighting and camera conditions to extract information. We believe such an attacker is within reason considering search and seizure procedures in many countries and states. However, a passive smudge attack, *e.g.*, via telephotography, can still be useful in a later active attack, where the touch screen device becomes available. The information obtained will often still be fresh – users tend to leave their passwords unchanged unless they suspect a compromise [3] – encouraging multiphase attack strategies.

3 Android Password Pattern

The Android password pattern is one of two unlock mechanisms, as of the release of Android 2.2 where alpha-numeric pins are now allowed [1]. However, the password pattern is currently the primary authentication mechanism on the vast majority of Android devices that have not yet received the update, and the pattern remains an authentication option on Android 2.2 devices.

The Android pattern is one style of graphical passwords where a user traverses an onscreen 3x3 grid of contacts points. A pattern can take on a number of shapes and can be defined as an ordered list of contact points (Fig. 1 provides an indexing scheme). For example, the "L" shaped password can be represented as the ordered list $|14789|$, *i.e.*, the user begins by touching contact point 1, drawing downward towards point 7, and finally across to point 9[3].

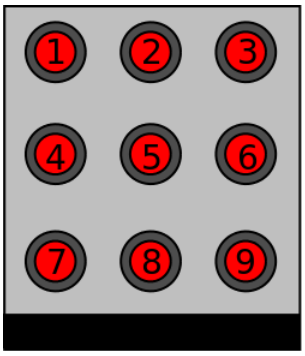

Figure 1: An illustration of the Android password pattern screen with overlaid identification numbers on contact points.

There are a three restrictions on acceptable patterns. It must contact a minimum of four points, so a single stroke is unacceptable. Additionally, a contact point can only be used once. These two restrictions imply that every pattern will have at least one direction change, and as the number of contact points increases, more and more such direction changes are required. Such convoluted connections of smudges may actually increase the contrast with background noise, as one of our experiments suggests (see Sec. 5).

The last, and most interesting, restriction applies to intermediate contact points: If there exists an intermediate point between two other contact points, it must also be a contact point, unless, that point was previously contacted. For example, in the "L" shaped pattern, it must always contain points 4 and 8 even though the ordered list $|179|$ would construct the exact same pattern If a user attempted to avoid touching either point 4 or 8, both would be automatically selected. Conversely, consider a "+" shaped pattern constructed by either the order list $|25846|$ or $|45628|$, the connected points $|46|$ or $|28|$ are allowed because point 5 was previously contacted.

Due to the intermediate contact point restriction, the password space of the Android password pattern contains 389,112 possible patterns[4]. This is significantly smaller than a general ordering of contact points, which contains nearly 1 million possible patterns. Still, this is a reasonably large space of patterns, but when considering information leakage of smudge attacks, an attacker can select a highly likely set of patterns, increasing her chances of guessing the correct one before the phone locks-out[5]. Sometimes, even, the precise pattern can be determined.

[3] Although a pattern can be entered using two fingers, stepping in order to simulate a drag from dot-to-dot, it is unlikely common practice because it requires more effort on the part of the user and is not part of the on-screen instructions provided by Android.

[4] Due to the complexity of the intermediate contact point restriction, we calculated this result via brute force methods.

[5] Android smartphones require the user to enter a Google user-name and password to authenticate after 20 failed pattern entry attempts.

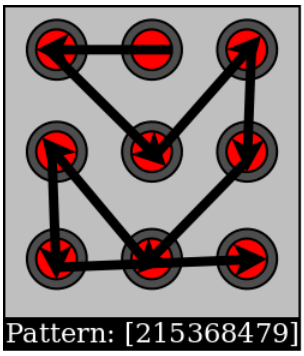

Figure 2: Password pattern used for captures; it contains streaks in all orientations and most directions.

4 Experimental Setup

In this section we present our experimental setup for capturing smudges from smartphone touch screens, including a background on photography and lighting. We experimented with two Android smartphones, the HTC G1 and the HTC Nexus1, under a variety of lighting and camera conditions. We also experimented with simulated phone application usage and smudge distortions caused by incidental clothing contact.

4.1 Photography and Lighting

This paper primarily investigates the camera angles and lighting conditions under which latent "smudge patterns" can be recovered from touchscreen devices. The fundamental principles of lighting and photographing objects of various shapes and reflective properties are well understood, being derived from optical physics and long practiced by artists and photographers. But the particular optical properties of smartphone touchscreens and the marks left behind on them are less well understood; we are aware of no comprehensive study or body of work that catalogs the conditions under which real-world smudges will or will not render well in photographs of such devices.

A comprehensive review of photographic lighting theory and practice is beyond the scope of this paper; an excellent tutorial can be found, for example, in [7]. What follows is a brief overview of the basic principles that underlie our experiments. In particular, we are concerned with several variables: the reflective properties of the screen and the smudge; the quality and location of the light sources; and finally, the location of the camera with respect to the screen.

Object surfaces react (or do not react) to light by either *reflecting* it or *diffusing* it. Reflective surfaces (such as mirrors) bounce light only at the complementary angle from which it arrived; an observer (or camera) sees reflected light only if it is positioned at the opposite angle. Diffuse surfaces, on the other hand, disperse light in all directions regardless of the angle at which it arrives; an observer will see diffused light at any position within a line of site to the object. The surfaces of most objects lie somewhere on a spectrum between being completely reflective and completely diffuse.

Lighting sources vary in the way they render an object's texture, depending on both the size and the angle of the light. The *angle* of the light with respect to the subject determines which surfaces of the object are highlighted and which fall in shadow. The *size* of the light with respect to the subject determines the range of angles that keep reflective surfaces in highlight and how shadows are defined. Small, point-size lights are also called *hard* lights; they render well-defined, crisp shadows. Larger light sources are said to be *soft*; they render shadows as gradients. Finally, the angle of the camera with respect to the subject surface determines the tonal balance between reflective and diffuse surfaces.

These standard principles are well understood. What is not well understood, however, is the reflective and diffuse properties of the screens used on smartphone devices or of the effects of finger smudges on these objects. We conducted experiments that varied the angle and size of lighting sources, and the camera angle, to determine the condition under which latent smudge patterns do and do not render photographically.

4.2 Photographic Setup

Our principle setup is presented in Fig. 3. We use a single light source (either soft, hard lighting, or omnidirectional lighting via a lighting tent) oriented vertically or horizontally. A vertical angle increments in plane with the camera, while a horizontal angle increments in a perpendicular plane to the camera. All angles are measured with respect to the smartphone.

Vertical angles were evaluated in 15 degree increments, inclusively between 15 and165 degrees. Degrees measures are complementary for vertical and lens angles. For example, a lens angle of 15 degrees and a vertical angle of 15 degrees are exactly complementary such that light reflects off the touch screen into the camera like a mirror. Horizontal angles were evaluated inclusively between 15 and 90 degrees as their complements produce identical effects. Similarly, we only consider camera angles between 15 and 90 degrees, inclusively; *e.g.*, a vertical and lens angle both at 105 degrees is equivalent to a vertical and lens angle both at 15 degrees with just the light and camera switch. Additionally, when the lens angle is at 90 degrees, only vertical lighting angles of 15 to 90 degrees need consideration[6]. Finally, for omnidirectional light only the lens angles need to be iterated as

[6]We do not consider 180 or 0 degree angles, which cannot provide lighting or exposure of the smudges.

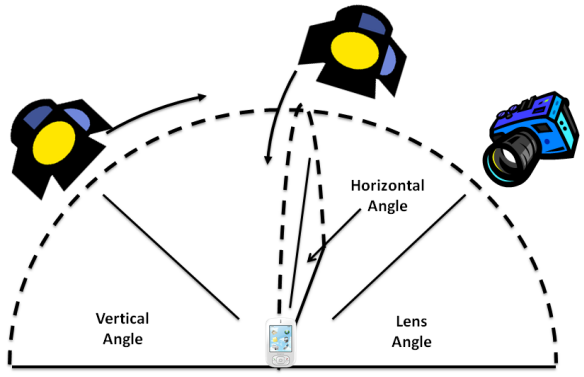

Figure 3: Principle Photographic Setup: The lighting and camera conditions at various vertical lighting angles (in plane with camera), horizontal lighting angles (in perpendicular plane with camera), and lens angles with respect to the smartphone.

light is dispersed such that it seems it is originating from all possible angles.

In total, there are 188 possible setups. For the base lighting condition, hard or soft, there are 11 vertical and 6 horizontal angles for 5 possible lens angles, not including the 90 degrees lens angle which only has 6 possible setups. With the addition of 6 lens angles for omnidirectional lighting, that leaves $188 = 2(5 \times 17 + 6) + 6$ setups, but there is still overlap. A 90 degree angle vertically and horizontally are equivalent, resulting in 178 unique setups.

4.3 Equipment Settings

We used relatively high end, precision cameras, lenses, lighting, and mounting equipment in our experiments to facilitate repeatability in our measurements. However, under real-world conditions, similar results could be obtained with much less elaborate (or expensive) equipment and in far less controlled environments.

All photographs were captured using a 24 megapixel Nikon D3x camera (at ISO 100 with 16 bit raw capture) with a tilting lens (to allow good focus across the entire touch screen plane). The camera was mounted on an Arca-Swiss C-1 precision geared tripod head. The large ("soft") light source was a 3 foot Kino-Flo fluorescent light panel; the small ("hard") light was a standard cinema "pepper" spotlight. For single light experiments, the directional light was at least 6 stops (64 times) brighter than ambient and reflected light sources. For omnidirectional lighting, we used a Wescott light tent, with light adjusted such that there was less than a 1 stop (2x) difference between the brightest and the dimmest light coming from any direction. All images were exposed based on an incident light reading taken at the screen surface.

4.4 Pattern Selection and Classification

In all experiments, we consider a single pattern for consistency, presented in Fig. 2. We choose this particular pattern because it encompasses all orientation and nearly all directions, with the exception of a vertical streak upwards. The direction and orientation of the pattern plays an important role in partial information collection. In certain cases, one direction or orientation is lost (see Sec. 6).

When determining the effectiveness of pattern identification from smudges, we use a simple classification scheme. First, two independent ratings are assigned on a scale from 0 to 2, where 0 implies that no pattern information is retrievable and 2 implies the entire pattern can be identified. When partial information about the pattern can be observed, *i.e.*, there is clearly a pattern present but not all parts are identifiable, a score of 1 is applied. Next, the two independent ratings are combined; we consider a pattern to be fully identifiable if it received a rating of 4, *i.e.*, both classifiers indicated full pattern extraction[7].

We also wished to consider the full extent of an attacker, so we allow our classifiers to adjust the photo in anyway possible. We found that with a minimal amount of effort, just by scaling the contrast slighting, a large number of previously obscured smudges become clear. Additionally, all the image alterations performed are equivalent to varying exposure or contrast settings on the camera when the image was captured.

5 Experiments

In this section, we present our experiments to test the feasibility of a smudge attack via photography. We conducted three experiments: The first considers ideal scenarios, where the touch screen is clean, and investigated the angles of light and camera that produce the best latent images. The results of the first experiment inform the later ones, where we simulate application usage and smudge removal based on contact with clothing.

5.1 Experiment 1: Ideal Collection

The goal of this experiment was to determine the conditions by which an attacker can extract patterns, and the best conditions, under ideal settings, for this. We consider various lighting and camera angles as well as different styles of light.

Setup. In this experiment we exhaust all possible lighting and camera angles. We consider hard and soft lighting as well as completely disperse, omnidirectional lighting, using a total of 188 photographs in classification. We

[7] We note that this rating system can lead to bias because the same pattern is used in every photograph. Specifically, there may be projection bias; knowing that a smudge streak is present, the classifier projects it even though it may not necessarily be identifiable. We use two independent classifiers in an attempt to alleviate this bias and only consider full pattern retrieval if bother classifiers rate with value 4.

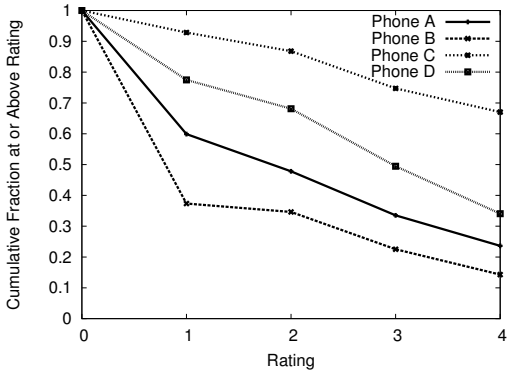

Figure 4: Cumulative Fraction Graph for Experiment 1: For each rating and phone, the cumulative fraction of photos scoring that rating, or higher.

App. Noise	G1 over	G1 under	Nexus 1 over	Nexus 1 under
dots	4	4	2.7	3.7
streaks	3	2	3	3
dots & steaks	3	1.6	4	3
face	4	2.3	4	2

Table 1: Results of Experiment 2: The average rating with application usage for patterns entered over and under the application noise.

experiment with four phones with different qualities of pattern entry, referred to by these letter identification:

- **Phone A:** HTC G1 phone with the pattern entered using "normal" touches
- **Phone B:** HTC G1 phone with the pattern entered using "light" touches
- **Phone C:** HTC G1 phone with the pattern entered after the phone has been held in contact with a face, as would happen after a phone call
- **Phone D:** HTC Nexus 1 phone with pattern entered using "normal" touches

Results. As described previously, each photograph is rated by the combination of two unique ratings on a scale from 0 to 2, which when combined provide a rating on a scale between 0 and 4. The key results of this classification are presented in Fig. 4 as a cumulative fraction graph.

The pattern that was most easily identifiable was Phone C, where the phone was first placed on a face prior to pattern entry. In roughly 96% of the photographic setups, a partial pattern was retrievable (*i.e.*, a rating of at least 1), and in 68% of the setups, the complete pattern was retrieved (*i.e.*, a rating of 4).

In contrast to the other tested phones, Phone C was dirty prior to password entry as broad smudging occurred due to contact with the facial skin. Entering the pattern on top of this broad smudge greatly contrasted with the pattern entry smudges (see Fig. A5). We explore this phenomenon further in Experiment 2. It is important to note that entering a pattern after a phone call is likely common because most conversations are longer than the phone lockout period. If a user wants access to other applications post hang-up, she will have to enter her pattern.

Phone B was the worst performing pattern entry. In this case, the pattern was entered using light touching, yet in over 30% of the setups, some partial information was retrievable. Moreover, in 14% of the photographs, the complete pattern is retrievable.

By far the best lens angle for retrieval was 60 degrees (followed closely by 45 degrees). In more than 80% of the lighting scenarios with a 60 degree lens, perfect or nearly perfect pattern retrieval was possible with a 60 degree camera angle. The worst retrieval was always when the vertical and lens angle were complimentary which transformed the touch screen surface into a mirror, effectively washing out the smudges (see Fig. A4 for one such example). Additionally, omnidirectional light (*i.e.*, using the light tent), had a similar effect. Omnidirectional light implies that there always exists a perfect reflection into the camera as light is emitted from all angles.

The most interesting observation made from the photographs is that in many of the setups, the *directionality* of the smudges can be easily discerned. That is, the order of the strokes can be learned, and consequently, the precise pattern can be determined. As an example see Fig. 5. At each direction change, a part of the previous stroke is overwritten by the current one, most regularly at contact points. On close inspection, the precise order of the contact points can be clearly determined and the pattern becomes trivially known.

5.2 Experiment 2: Simulated Usage

In this experiment, we were interested in the affect that user applications have on the capabilities of an attacker. Previously, we demonstrated that talking on the phone may increase the contrast between a pattern smudge and the background; we further elaborate on that point here. Additionally, we investigate the affect of application usage as it may occur prior to or post pattern entry.

Setup. The setup of this experiment was informed by the results of the previous. We photographed the phones

Figure 5: Phone A, from Experiment 1, where the pattern is entered with normal touches. Notice that the directionality of the pattern can be determined at ever direction change.

Figure 6: Phone from Experiment 3, where the phone was wiped, placed (and replaced) in a pocket. Unlike Phone A from Fig. 5, some directionality is lost in the upper left portion of the pattern.

at a 45 degree lens angle and at three of the best vertical angles: 15, 75, and 90 degrees[8].

We based our usage simulation on the telephone application; an application installed on all Android smartphones. Although the phone application is not representative of all application usage, it has some important characteristics present in nearly all applications. If a user were to enter a phone number it would require a sequence of presses, or *smudge dots*, on the screen. A user could also scroll her contact list, causing up and down *smudge streaking*, or left and right, depending on the phones current orientation. Additionally, there may be combinations of these.

For each phone in the experiment – two G1 phones and two Nexus 1 phones – we consider 3 usage scenarios; (1) A grid of smudge dotting; (2) a hashing of up and down and left right smudge streaks; and (3), a combination of smudge dots and streaks in a grid and hash, respectively. We also consider if the pattern was entered prior to application usage (*i.e.*, the pattern is first entered, and then the steaks and/or dots are applied) or post application usage (*i.e.*, first steaks and/or dots are applied followed by the pattern). Finally, as a follow up to Experiment 1, we consider the effect of placing the touch screen surface to the face, before and after pattern entry. In all pattern entries, we assume normal touching.

Results. As before, each photograph was classified by two individuals, and the combined results are considered. The results are summarized in Table 1. In general, entering the pattern over the usage smudges is more clearly retrieved, as expected. Dots also tend to have less of an effect then streaks, again as expected.

Interestingly, the over pattern entry for the combination of dots and streaks on the Nexus 1 scored perfect retrieval (see Fig. A1 for a sample image). Upon closer inspection, this is due to the intricacy of the pattern – the many hooks and turns required for such a long pattern – created great contrast with usage noise, and thus the pattern was more easily retrieved. Finally, as expected based on the results of Experiment 1, broad smudging on the face provided perfect retrieval for the over case, and even in the under case, partial information was retrieved.

[8]Although 60 degrees lens angle performed best overall, the setup required for 45 degrees was much simpler and had similarly good results at these vertical lighting angles.

5.3 Experiment 3: Removing Smudges

In this experiment we investigated the effects of smudge distortion caused by incidental contact with or wiping on clothing.

Setup. Using the same photographic setup as in Experiment 2, we photographed two clothing interference scenarios, both including placing and replacing the phone in a jeans pocket. In the first scenario, the user first intentionally wipes the phone, places it in her pocket, and removes it. In scenario two, the user places the phone in her pocket, sits down, stands up, and removes it.

Although this does not perfectly simulate the effects of clothing contact, it does provide some insight into the tenacity of a smudge on a touch screen. Clearly, a user can forcefully wipe down her phone until the smudge is no longer present, and such scenarios are uninteresting. Thus, we consider incidental distortion.

Results. Surprisingly, in all cases the smudge was classified as perfectly retrievable. Simple clothing contact does not play a large role in removing smudges. However, on closer inspection, information was being lost. The directionality of the smudge often could no longer be determine (See Fig. 6 for an example). Incidental wiping disturbed the subtle smudge overwrites that informed directionality. Even in such situations, an attacker has greatly reduced the likely pattern space to 2; the pattern in the forward and reverse direction.

Figure 7: Phone from Experiment 1: One stroke of the pattern, $|84|$, is lost due to the camera or lighting angle. The contrast has been adjusted.

Figure 8: Phone from Experiment 2: With this usage condition (dot and streaks, under), the pattern is nearly all lost. The contrast has been adjusted.

5.4 Summary

Our photographic experiments suggest that a clean touchscreen surface is primarily, but not entirely, reflective, while a smudge is primarily, but not entirely, diffuse. We found that virtually any directional lighting source that is not positioned exactly at a complementary angle to the camera will render a recoverable image of the smudge. Very little photo adjustment is required to view the pattern, but images generally rendered best when the photo capture was overexposed by two to three f-stops (4 to 8 times "correct" exposure).

If the effect of the smudge is to make a chiefly reflective surface more diffuse, we would expect completely even omnidirectional light to result in very poor rendering of the image. And indeed, our experiments confirm this – even extensive contrast and color adjustment was generally unable to recover the smudge pattern from images captured under omnidirectional light under the light tent. Fortunately for the attacker, however, most "real world" lighting is primarily directional. The main problem for an attacker who wishes to surreptitiously capture a smudge pattern is not application noise or incidental clothing contact (as Experiment 2 and 3 showed) but rather ensuring that the angle of the camera with respect to the screen surface is not at an angle complementary to any strong light source.

6 Directions for Exploitation

We have demonstrated the ability of an attacker to capture information from smudges via photography. We now discuss how the information gained can be use to defeat the Android password pattern. As presented in Sec. 3, the size of the pattern space contains 389,112 distinct patterns. A significant number of those patterns can be eliminated as possible passwords by a smudge attacker. For example, perfect pattern retrieval with directionality is possible, reducing the possibilities to 1. Partial retrieval of patterns from smudges requires deeper analysis, towards which we present initial thoughts on exploiting captured smudges. Smudge data can be combined with statistical data on human behavior such as pattern usage distributions for large sets of users to produce likely sets of patterns for a particular smudged phone.

6.1 Using Partial Information

Using the photographs taken during our experiments, we investigated what was lost in partial retrieval scenarios. Two cases emerged: First, a lack of finger pressure and/or obscuration of regions of the photograph led to information loss. For example, in Fig. 7, the diagonal for connection $|48|$ cannot be retrieved. This partial retrieval is still extremely encouraging for an attacker, who has learned a good deal about which patterns are likely, e.g., it could be each isolated part uniquely, the two parts connected, etc.

Another case emerges when a significant amount of usage noise obscures the smudge present; e.g., Fig. 8 is a photo from Experiment 2 with dots and streaks over the pattern entry. An attacker may guess that two sets of "V" style diagonals are present, but in general the entire pattern is not observable. Moreover, using this information is not likely to reduce the pattern space below the threshold of 20 guesses.

However, an attacker may have access to many images of the same pattern captured at different points in time. By combining this information, it may be possible for an attacker to recreate the complete pattern by data fusion [6]. As an example, consider an attacker combining the knowledge gained from the Fig. 7 and Fig. 8; if it was known that the same pattern was entered, the bottom "V" shape in Fig. 8 is enough information to finish the pattern in Fig. 7.

6.2 Human Factors

Human behavior in password selection has been well studied [11, 15], and even in the graphical password space, password attack dictionaries based on human mnemonics or pattern symmetry have been proposed [17, 18]. Similar behaviors seem likely to emerge in the Android password space, greatly assisting an attacker.

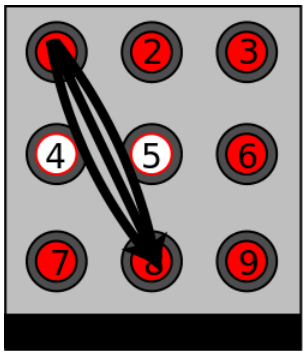

Figure 9: A 30 degree pattern stroke that is difficult to enter when points 4 and/or 5 are previously unselected.

We conjecture that the ease of pattern entry is an important factor for consideration because a user must enter her password on every phone lock; generally, a 30 second timeout. If the password is difficult to enter consistently, then it would be less usable and therefor less likely the user's chosen pattern. For example, the contact point stroke in Fig. 9 contains a 30 degree strokes which is prone to error when the intermittent contact points are not previously touched (*e.g.*, point 4 and 5). When considering this additional restriction, the password space can be reduced by over 50% to 158,410 likely patterns.

Another usability factor is pattern length: The longer the pattern, the longer the amount of time it takes to enter it. A frequent smartphone user may avoid excessively long patterns. In the same vein, a user may avoid frequent direction changes, as each requires, again, more time to enter. Other human factors may play a role in pattern selection and investigating them in the context of smudge attacks is an area of future research.

7 Related Work

Previous work on this subject is limited. Perhaps the closest related work was performed by Laxton *et al.* regarding copying physical keys based on photographic analysis [12], a so called *teleduplication attack*. An attacker may reproduce the key by measuring the position and relative depth of the key cuts to extract the bitting code. Once known, the key may be duplicated without the attacker ever having possessed it.

Smudge and teleduplication attacks are similar in a number of ways. First, both take advantage of visual information, whether password smudges or key bittings. Many of the same basic principles of teleduplication attacks, such as the photographic capturing methods, are relevant to our work. Both attacks are executed in physical space and can be done from afar. Finally, the usefulness of information gained requires next steps. In the case of a teleduplication attack, duplicating the key is only useful if the door it opens is known. In the same way, learning the pattern is only useful if the touch screen device were to come into the attacker's possession.

There has been a fair amount of interesting research performed on graphical passwords [2, 16]. Specifically, it should also be noted that there are several other proposed graphical password schemes [2, 4, 10]. We believe that several of these authentication procedures, if performed on a touch screen, may be susceptible to smudge attacks.

If smudge attacks were to be automated, previous work in the area automated image recognition, *e.g.* facial recognition techniques [5, 9] or optical character recognition [8, 13, 14], would be applicable. Such automated techniques are especially dangerous if an attacker possessed many successive images (*e.g.*, via video surveillance).

8 Conclusion

In this paper we explored *smudge attacks* using residual oils on touch screen devices. We investigated the feasibility of capturing such smudges, focusing on its effect on the password pattern of Android smartphones. Using photographs taken under a variety of lighting and camera positions, we showed that in many situations full or partial pattern recovery is possible, even with smudge "noise" from simulated application usage or distortion caused by incidental clothing contact. We have also outlined how an attacker could use the information gained from a smudge attack to improve the likelihood of guessing a user's patterns.

Next steps in our investigation include a deeper analysis of the advantages derived from analysis of smudges and an end-to-end smudge attack experiment on a small (voluntary) population of Android users who employ the password pattern. Additionally, we would like to perform a broad study of human factors in pattern selection as they relate to the Android pattern.

We believe smudge attacks based on reflective properties of oily residues are but one possible attack vector on touch screens. In future work, we intend to investigate other devices that may be susceptible, and varied smudge attack styles, such as heat trails [19] caused by the heat transfer of a finger touching a screen.

The practice of entering sensitive information via touch screens needs careful analysis in light of our results. The Android password pattern, in particular, should be strengthened.

Acknowledgments

We would like to thank the anonymous reviewers for their useful insight and thoughtful remarks, and Samual Panzer for his assistance during the experiments. Finally, this work was supported by *Biomedical Informa-*

tion Red Team (*BIRT*) NFS Award CNS-0716552 and *Security Services in Open Telecommunication Networks* NFS grant CNS-0905434.

References

[1] Android 2.2 platform highlights. http://developer.android.com/sdk/android-2.2-highlights.html.

[2] D. Davis, F. Monrose, and M. K. Reiter. On user choice in graphical password schemes. In *USENIX Sec'04*, 2004.

[3] A. M. DeAlvare. A framework for password selection. In *UNIX Security Workshop II*, 1998.

[4] H. Gao, X. Guo, X. Chen, L. Wang, and X. Liu. Yagp: Yet another graphical password strategy. *Computer Security Applications Conference, Annual*, 0:121–129, 2008.

[5] S. Gutta, J. R. Huang, H. Wechsler, and B. Takacs. Automated face recognition. volume 2938, pages 20–30. SPIE, 1997.

[6] D. L. Hall and J. Llinas. An introduction to multisensory data fusion. *Proc. IEEE*, 85(1), January 1997.

[7] F. Hunter and P. Fuqua. *Light: Science and Magic: An Introduction to Photographic Lighting*. Focal Press, 1997.

[8] S. Impedovo, L. Ottaviano, and S. Occhinegro. Optical character recognition – a survey. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 5(1-2):1–24, 1991.

[9] R. Jenkins and A. Burton. 100% accuracy in automatic face recognition. *Science*, 319(5862):435, January 2008.

[10] I. Jermyn, A. Mayer, F. Monrose, M. K. Reiter, and A. D. Rubin. The design and analysis of graphical passwords. In *USENIX Sec'99*, pages 1–1, Berkeley, CA, USA, 1999. USENIX Association.

[11] D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *USENIX Sec'90*, 1990.

[12] B. Laxton, K. Wang, and S. Savage. Reconsidering physical key secrecy: Teleduplication via optical decoding. In *CCS*, October 2008.

[13] J. Mantas. An overview of character recognition methodologies. *Pattern Recognition*, 19(6):425–430, 1986.

[14] S. Mori, H. Nishida, and H. Yamada. *Optical Character Recognition*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[15] R. Morris and K. Thompson. Password security: a case history. *Communnincations of the ACM*, 22(11):594–597, 1979.

[16] K. Renaud and A. D. Angeli. Visual passwords: Cure-all or snake-oil. *Communications of the ACM*, 52(12):135–140, December 2009.

[17] J. Thorpe and P. van Oorschot. Graphical dictionaries and the memorable sapce of graphical passwords. In *USENIX Sec'04*, August 2004.

[18] J. Thorpe and P. C. van Oorschot. Human-seeded attacks and exploiting hot-spots in graphical passwords. In *USENIX Sec'07*, 2007.

[19] M. Zalewski. Cracking safes with thermal imaging, 2005. http://lcamtuf.coredump.cx/tsafe/.

A Appendix

Figure A1: A phone from Experiment 2: The pattern contrasts greatly with the background noise; a grid of dots. The contrast on this image has been adjusted.

Figure A2: An image from Experiment 1: All four phones clearly displayed the pattern without the need to adjust contrast. Even the lightly touched Phone B has a visible pattern.

Figure A3: An image from Experiment 2: Even with background noise (over, on the left, and under, on the right of the pattern entry), either partial or complete pattern identification is possible as it contrast with such usage noise. The contrast on these images has been adjusted.

Figure A5: Phone C from Experiment 1: Without any image adjustment, the pattern is clear. Notice that the smudging, in effect, cleans the screen when compared to the broad smudging caused by facial contact. This contrast aids in pattern identification, as demonstrated in Experiment 2.

Figure A4: An image from Experiment 1: Complimentary lighting and lens angle causes significant glare, leading to unidentifiable patterns and information loss.

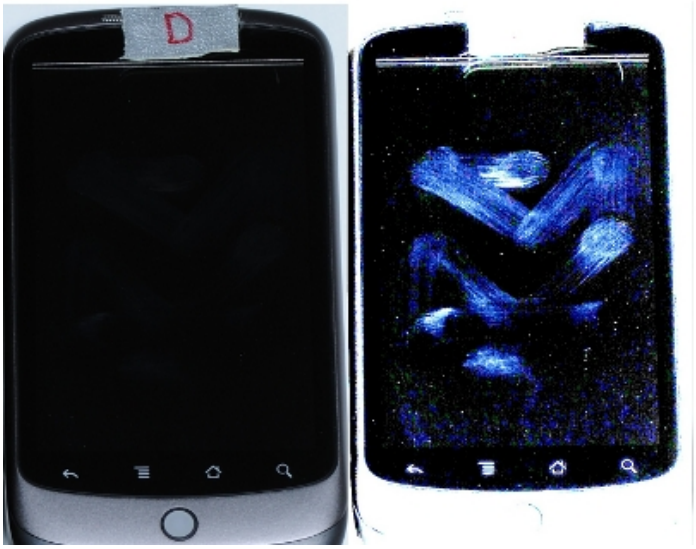

Figure A6: Phone D from Experiment 1, prior to and post contrast adjustment: In many situations, adjusting the levels of color or contrast can highlight a smudge previously obscured. The images on the left and right are identical.

Privacy-Aware Message Exchanges for Geographically Routed Human Movement Networks

Adam J. Aviv[+], Matthew Blaze[+], Jonathan M. Smith[+] and Micah Sherr[*]
[+] University of Pennsylvania [*] Georgetown University

ABSTRACT

This paper introduces a novel privacy-aware geographic routing protocol for *Human Movement Networks* (HumaNets). HumaNets are fully decentralized opportunistic store-and-forward, delay-tolerate networks composed of smartphone devices. Such networks allow participants to exchange messages *phone-to-phone* and have applications where traditional infrastructure is unavailable (*e.g.*, during a disaster) and in totalitarian states where cellular network monitoring and censorship are employed.

Our protocol leverages self-determined *location profiles* of smartphone operators' movements as a predictor of future locations, enabling efficient geographic routing. Since these profiles contain sensitive information about participants' *prior movements*, our routing protocol is designed to minimize the exposure of sensitive information during a message exchange. We demonstrate via simulation over both synthetic and real-world trace data that our protocol is highly scalable, provides reasonable performance, and leaks little information.

1. INTRODUCTION

The ubiquity of smartphones enable new communication models beyond those provided by cellular carriers. While standard cellular communication uses a centralized infrastructure that is maintained by the service provider, smartphones have communication interfaces such as ad-hoc WiFi and Bluetooth that allow direct communication between devices. Since smartphone owners often carry their devices and encounter other individuals (and their smartphones) in their daily routines, *smartphones enable fully decentralized store-and-forward networks that completely avoid the cellular infrastructure*.

Human Movement Networks (HumaNets) [3] fit this model and are designed to allow participants to exchange messages phone-to-phone without using any centralized infrastructure. HumaNets' "out-of-band" message passing is applicable when cellular networks are unavailable or if the networks are untrusted (*i.e.*, operated by a totalitarian state that censors [14], shuts down [36], or otherwise leverages its communication systems to restrict its citizenry [17]).

Rather than rely on network addresses, HumaNets route messages using *geocast* – an addressing scheme that directs messages towards a particular geographic region. To cope with mobility, HumaNet routing protocols route messages based on message carriers' predicted *future* locations. This is accomplished by leveraging self-determined *location profiles* that approximate the smartphone owners' routine movements. The patterns of human mobility – for example, the daily commute to and from work – serve as predictors of future locations. HumaNets take advantage of this observation by greedily forwarding messages to smartphones whose owners' location profiles indicate that they are good candidates for delivery.

Unfortunately, many existing routing protocols that have been developed in the context of MANETs or DTNs are incompatible with HumaNets due to either their lack of support for the *dynamism* of smartphone networks or their inability to safeguard sensitive location information. Existing routing approaches often assume highly connected and mostly static networks [11, 33, 37, 42, 43]. Additionally, privacy issues must be central when designing a HumaNet routing protocol since location profiles contain sensitive information about participants' *prior movements*. The disclosure of such information is particularly dangerous when HumaNets are used for covert communication in totalitarian regimes. Prior approaches that do not consider privacy [18, 20], rely on trusted third parties [13], or assume *a priori* trust relationships [6] are also unsuitable for HumaNets.

This paper proposes a novel routing protocol for HumaNets that protects participants' location profiles from an adversary who wishes to learn previous movements and/or determine "important" locations of network users (*e.g.*, home, work, or the location of underground activist meetings). Our technique, which we call *Probabilistic Profile-Based Routing* (PPBR), balances performance and privacy by efficiently routing messages in a manner that minimizes the exposure of users' location profiles. We demonstrate through trace-driven simulations using both real-world and synthetic human movement data that our PPBR protocol is highly scalable, efficiently routes messages, and preserves the privacy of profile information. In summary, the contributions of this paper are:

- The design and introduction of a fully decentralized, privacy-preserving, geographic-based HumaNet message routing protocol for smartphones;
- An analysis of the privacy and security properties offered by our routing protocol; and
- A trace-driven simulation study (using both real-world and synthetic data) that evaluates our method's scalability and efficiency.

2. NETWORK ASSUMPTIONS & GOALS

To achieve reasonable performance, HumaNets leverage humans' tendency to follow *routines*: The locations that people frequented in the past are predictors of their future locations [3]. However, a device's location history may be extremely sensitive, and moreover, combining multiple nodes'

location histories may allow an adversary to discover social networks and enumerate participants' movements. Hence, the high-level goal of our PPBR protocol and the central challenge of this paper is to enable *efficient geographic -based messaging that limits the exposure of information at message exchanges*. In particular, an adversary who witnesses a message exchange would learn little *important* information about the participants' location histories.

Importantly, however, our HumaNet routing protocol does not conceal the identities of the network's participants. An adversary who intercepts a PPBR message can reasonably conclude that the sender is participating in a HumaNet. Participating in a HumaNet inherently carries risk if used as an anti-censorship technology: This is unfortunately true of any system that may be deemed "subversive". However, when other means of communication are impossible (either due to global monitoring or blocked connectivity), HumaNets provide a *means* to exchange information in a manner that is efficient, scalable, difficult to surveil, and privacy-aware.

Requirements. HumaNets routing protocols are designed for location-aware mobile devices. We assume that network participants can learn their locations (*e.g.*, via GPS[1]) without relying on the cellular service provider's network, and that devices contain sufficient storage to record their movement histories.

We additionally assume that participants have knowledge of the routing area. Since HumaNets enable geocast routing, a message that is targeted at specific receivers requires the sender to have some knowledge about the receivers' likely future locations (*e.g.*, their home or work); this requirement is similar to that imposed by traditional networking where users need knowledge of a service's hostname or IP address. We also assume that participants know some coarse-grain information about general movement statistics over the routing area. In particular, nodes should be capable of estimating the "popularity" of city areas – *e.g.*, that the upper west side of Manhattan is more densely traveled than Far Rockaway, Queens. This information can be obtained from census data, other public source of information, or personal experience. Such information can be shipped with the HumaNets software and is assumed to be known to an adversary.

Threat Model. We envision both passive and active adversaries. A passive adversary may have any number of confederates and is able to observe message exchanges at a fixed number of locations throughout the HumaNet routing area. An active adversary may additionally participate in HumaNets by generating fake messages, accepting messages, and/or dropping or misrouting messages.

We do not provide protection against a *mobile targeting adversary*. An adversary that can physically follow a node can trivially learn about its whereabouts and discover its routine movements. Such a "stalker" adversary is also very costly to deploy. In this paper, we focus on less targeted attackers and assume an adversary who monitors, intercepts, or participates in local exchanges that occur in its presence. The adversary is thus aware of the participants and their locations at the time of an exchange, and we do not claim that our system provides traditional location-privacy [19] for ad hoc networks, although such extensions may be relevant here. As such, the adversary's goals are as follows:

- DISRUPTION: Inject failures into the network such that messages can no longer be reliably delivered.
- DE-ANONYMIZATION: Determine the originating sender of intercepted messages.
- PROFILING: Infer movement patterns of a targeted individual or learn his/her "important" locations (*e.g.*, home, work, underground meeting place).

Performance and Security Goals. The goal of our routing protocol is to provide the following properties in the presence of active and passive adversaries:

- RELIABILITY: Messages should reach their intended destinations with high probability.
- EFFICIENCY: Messages should reach their intended destinations with reasonable latency and overhead.
- SCALABILITY: HumaNets should be able to scale to a large number of participants with many concurrent messages.
- POINT-TO-POINT: Messages should be exchanged only point-to-point and avoid any centralized routing structures.
- PRIVACY-PRESERVATION: The protocol should not leak the sender's identity, nor should it reveal information about participants' previous locations.

At first blush, it may seem that naïve flooding and random walk strategies are sufficient to achieve the above goals. Although these strategies achieve the POINT-TO-POINT and PRIVACY-PRESERVATION properties, they are lacking with respect to SCALABILITY, EFFICIENCY, and/or RELIABILITY. In particular, flooding achieves optimal latency and delivery rates because all paths are explored, but scales poorly since all transfers that do not occur along the optimal path constitute a wasted effort (and, consequently, wasteful power consumption). Moreover, since several senders may use HumaNets to disseminate their messages, flooding requires that nodes store (and worse, communicate) a large fraction of all messages. At the other extreme, random walk protocols in which messages are transferred (as opposed to copied) upon node contacts scales well but incurs poor RELIABILITY and EFFICIENCY.

It may also seem that traditional cryptographic solutions would be applicable here. However, the decentralized and highly dynamic nature of HumaNets make their deployment difficult. In particular, many cryptographic solutions require centralized services or trusted third parties. Such approaches are problematic in our setting since a strong (*e.g.*, nation-state) adversary could either compromise or prevent access

[1]GPS is a unidirectional protocol and requires only the reception of signals from U.S.-operated satellites.

to centralized services. Routing techniques that rely on complex key distribution schemes or expensive cryptographic operations (for example, SMC [44]) are incompatible with HumaNets' distributed architecture and use of power-constrained devices. A significant advantage of PPBR is that it provides PRIVACY-PRESERVATION using simple probabilistic techniques, and avoids the key management and computation issues present in protocols that provide more traditional cryptographic protections [6, 13, 38].

3. PRIVACY-PRESERVING ROUTING

At a high level, the *Probabilistic Profile-Based Routing* (PPBR) protocol requires participants (nodes) to *estimate* whether they are good candidates for delivering a message. Upon receiving a message from a *carrier — i.e.*, a node that announces a message — the receiving node makes a local determination as to whether it is well positioned to deliver the message to the addressed destination. The node either *accepts* or *discards* the message, and in either case, *does not notify the current carrier as to its choice*. If the message is accepted, the receiving node becomes a carrier and begins to announce the message. However, unlike flooding techniques in which messages are continuously duplicated leading to an exponential number of message copies, each message carrier in PPBR announces the message to only k contacts, of which only one out of the k receiving nodes should accept it. The main task is thus for a receiver to locally determine whether it is best suited to deliver the message out of the $k-1$ other nodes that received the message.

3.1 HumaNet Preliminaries

Addressing. HumaNets provide a basic addressing primitive, *geocast*, in which messages are addressed to a geographic location (*e.g.*, a city square). Messages are routed to nodes who are likely to travel towards the destination address and are then locally flooded within the confines of the specified destination. We do not consider temporal features in addressing or routing – *i.e.*, addressing a message to a location for a specific time – but the protocol described herein can be easily expanded to meet temporal specifications[2]. Additionally, HumaNets do not provide message confidentiality; however, message payloads can be protected using standard encryption techniques.

HumaNets interpret the routing area as a grid, the dimensions of which are assumed to be known *a priori* to all nodes (for example, based on latitude and longitude). Messages are addressed to a particular grid square. In the remainder of the paper, when describing a message address or destination, we refer to the index of the corresponding grid square.

[2]One method is for nodes to maintain multiple location profiles, each representing movement information collected at different times of the day. The message exchange algorithm is as described later; however, each node now uses the location profile most relevant to the addressed time and location. With this addition, a message carrier is not only likely to deliver the message to the location, but also deliver it at the specified time.

Message Exchanges. Messages are exchanged between smartphone devices when they come into wireless contact with one another. We consider a contact to occur when two nodes are within wireless transmission range, *e.g.*, the range of Bluetooth or a point-to-point 802.11 transmission in ad hoc mode. At set time intervals, nodes awaken and begin the routing protocol. If a contact is made, messages can be exchanged. Otherwise, if there are no other participants nearby, the node returns to normal activity.

HumaNets require coarse time synchronization (*i.e.*, within a few seconds) to ensure message exchanges occur at the appropriate times. Such synchronicity could be achieved using NTP servers, but this would require nodes to send messages over centralized networks. Fortunately, smartphone devices are already highly synchronized as a requirement of participating in the centralized cellular network [2, 32] (a network which HumaNets do not use to send messages). If cellular services are disabled or are untrusted to provide correct time information, nodes could alternatively obtain the timing information from GPS satellite timestamps.

3.2 Routing Overview and Constructions

PPBR consists of two phases: a *passing phase* and a *holding phase* (see Figure 1). In the passing phase, a carrier of a message attempts to pass the message to the first k nodes that it encounters. A node that receives a message will locally estimate whether it has the highest similarity to the message address (a grid square) out of the $k-1$ other nodes who also received (or will receive) the message. If the node perceives itself to be the best candidate for delivery, it accepts the message, becomes a carrier, and prepares to transition to the passing phase. Otherwise, the message is dropped. A node transitions from the passing phase to the holding phase once it has announced the message to k other neighbors.

The challenge of PPBR is enabling each node to accurately predict whether it is the best of k candidates to accept a message *without conferring with other nodes*. The intuition behind our approach is that a node can compute a *similarity score* to a message's destination using its *location profile* – a compact representation of its movement history. To populate its location profile, a node periodically records its GPS location and determines the fraction of time spent within each grid square. Using its location profile along with background knowledge of the movement patterns of an "average" node, the node can estimate how well it is positioned to deliver the message relative to the $k-1$ other participants who will receive the message.

An important characteristic of PPBR's passing phase is that message reception is not acknowledged. An eavesdropper therefore cannot determine whether a message was accepted or declined by a nearby node. This makes it difficult for an adversary to conduct PROFILING attacks against a receiver, since it has no information to form a judgment as to whether the receiver's profile is well-suited for delivering the message. (We explore the effectiveness of PROFILING attacks against a carrier who announces a message

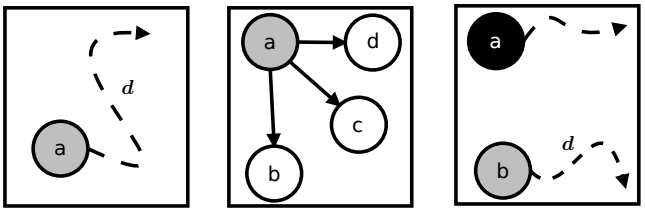

Figure 1: Overview of PPBR routing. (1) The initial message carrier (node a) enters the passing phase (grey shading). (2) The carrier encounters three nodes. (3) Node b considers itself the best of k candidates an accepts the message, becoming a carrier and initiating its passing phase. After advertising k messages, node a enters the holding phase (black shading).

in Section 5.) To further aggravate PROFILING attacks, if a node accepts a message and becomes a carrier, it does not announce the message until it has moved a distance d away from its current location, preventing the eavesdropper from observing the transition.

After a carrier has performed k message announcements, it transitions to the holding phase. In the holding phase, the carrier maintains the message for some time period, during which the node, hopefully, enters the message's addressed grid square and starts the local flood (restricted to the destination grid square). If the node does not reach the addressed grid square within a *local timeout*, the carrier drops the message. A message also has an associated *global timeout* after which all carriers drop the message.

Location Profiles. Nodes compute *location profiles* based on their movement histories.[3] Although long term collection could be useful in constructing a profile, HumaNets rely on shorter historical windows to minimize the effects from non-repeated movements, *e.g.*, vacations.

Each node periodically polls its location (*e.g.*, via GPS) to update its location profile. The profile is a matrix indexed by geographic grid square such that the value at position $\langle x, y\rangle$ is the normalized number of location readings in which the node was located at position $\langle x, y\rangle$ in the grid. That is, the value at position $\langle x, y\rangle$ in the location profile corresponds to the frequency that the node visited location $\langle x, y\rangle$ in the physical world over some time window. Following our heuristic, we assume that the matrix value at $\langle x, y\rangle$ (which is defined based on past behavior) approximates the node's future likelihood of visiting location $\langle x, y\rangle$ in the physical topology.

More formally, consider a current window of location entries $W = (\langle x_i, y_i\rangle, \langle x_j, y_j\rangle \ldots)$ that are already mapped to grid square references. The profile p, indexed by grid squares, contains the values:

$$p[\langle x, y\rangle] = \begin{cases} \frac{|W_{\langle x,y\rangle}|}{|W|} & \text{if } \langle x, y\rangle \in W \\ 0 & \text{otherwise} \end{cases} , \qquad (1)$$

[3]Recent revelations suggest that popular smartphones may already collect and store such information [5].

where $W_{\langle x,y\rangle}$ is the sub-list containing location entries occurring within the grid square $\langle x, y\rangle$, $p[\cdot]$ is the index function returning the associated value, and $|\cdot|$ indicates the length of the list.

General Node Profile. An advantage of PPBR is that it does not require nodes to share their location profiles. However, the technique assumes some globally shared information which we call the *general node profile*. The general node profile is a model of the "average" node's movement, and has the same structure and features as the standard location profile. Rather than representing the frequented locations of a single node, the general profile expresses the patterns of the general population. We assume that the general node profile is included with HumaNet software.

As we demonstrate in Section 4, the general node profile does not have to be a perfect model, and can be based on a rough estimate of population densities. In practice, we posit that a sufficient general node profile could be constructed using public data such as population densities from census data, transportation studies [41], or common knowledge.

Marginal Similarity. A node determines if it is the best of $k-1$ other message recipients by comparing its similarity with the message's destination to the "average" node's similarity calculated using the general node profile. If the node's similarity is a factor greater, the message is accepted.

More precisely, a node must first be able to calculate the similarity of a location profile to a message address (grid square). This is done by considering not only the value in the profile at the addressed grid-point, but also the values at nearby grid-points, discounted by their square distance. Formally, we define the similarity of a node n to a message m addressed to a_m to be:

$$\mathsf{sim}(p, a_m) = p[a_m] + \sum_{\substack{a_p \in p \\ a_p \neq a_m}} \frac{p[a_p]}{\mathsf{dist}(a_p, a_m)^2} , \qquad (2)$$

where p is a location profile and $\mathsf{dist}(a_p, a_m)$ denotes the Euclidean distance between grid-points a_p and a_m. This computation captures the desired property that a node that more frequently visits the message's targeted destination (and nearby areas) will have higher similarity than a node that visits the destination region less often[4].

A similarity score computed with the general node profile, rather than an individual node's profile, represents an estimate of the "average" node's similarity to the message address. We define the relationship between a node n's similarity and that of the general node's similarity as the *marginal similarity* σ. It is calculated as $\sigma = \frac{\mathsf{sim}(p_n, a_m)}{\mathsf{sim}(p_g, a_m)}$, where p_n is the profile of node n and p_g is the general node profile. The

[4]In our simulations, we found that a squared decay function (*i.e.*, the importance of similarity decreases as the square of the distance from the message address) produces good results. We have additionally experimented with other decay functions, and found that they produce similar (but slightly degraded) performance.

marginal similarity speaks to how well a node is suited to become a carrier of a message addressed to a_m as compared to a node on average: higher values indicate the node would make a good message carrier, while lower values indicate a poor carrier. The next challenge is selecting a threshold value for σ at which point only one of the k nodes that received the message will accept it and become a carrier.

Threshold Selection. We define τ as the *threshold marginal similarity score* at which a node accepts a message and becomes a carrier. Intuitively, τ should be the marginal similarity such that $1/k$ marginal similarity calculations are greater than τ. The threshold is calculated locally (and privately) by each node. First, a node computes σ for every grid square in p_g:

$$\bar{\sigma} = \left\langle \frac{\mathsf{sim}(p_n, a)}{\mathsf{sim}(p_g, a)} \;\middle|\; \forall\, a \in p_g \right\rangle \tag{3}$$

The computations are arranged in a sorted list $\bar{\sigma}$, where $\bar{\sigma}_i < \bar{\sigma}_j$ if $i < j$. $\bar{\sigma}$ represents marginal similarity calculations for all likely message addresses, and we wish the node to accept a message for $1/k$ of those addresses. To do this, a node chooses τ such that $1/k$ values in $\bar{\sigma}$ are greater than τ; more precisely, $\tau = \bar{\sigma}_i$ and $i = \lfloor |\bar{\sigma}| * (k-1)/k \rfloor$, where $|\cdot|$ denotes the length function. τ must be updated whenever the node's location profile changes. To conserve battery, such a computation could occur nightly while the device is charging.

In summary, PPBR messaging supports geocast: Messages are addressed to a particular grid square and intended for all participants residing therein. A message carrying node (a carrier) in the passing phase will duplicate the message to k other nodes before transitioning to the holding phase. Of the k nodes that receive a message, $k-1$ should drop the message while a single node should retain it. This process is oblivious to the message sender (and an adversary) who is unaware of which of the nodes accepted the message and which dropped it. To determine if a node is a good carrier (*i.e.*, the best of k), a receiving node computes their marginal similarity σ, which compares their similarity to that of the general node's, as embodied by the general node profile. If σ is greater than their locally calculated threshold τ, the message is accepted, otherwise it is rejected. Nodes that accept a message will transition to a passing phase after traveling a distance d from the point of reception, where they repeat the process by exchanging the message with k other nodes. At any point, the message may reach the addressed grid square, within which, the message is flooded to all participants present. Additionally, if a node does not deliver a message within a local timeout, the message is dropped. After a global timeout occurs, all message copies in the network are discarded.

4. PERFORMANCE EVALUATION

To evaluate the performance of PPBR, we constructed a discrete event-driven HumaNets simulator. Our simulator takes as input a trace of human (cellphone) movement and overlays the PPBR routing algorithm. In all simulations, we choose k to be 10 and conduct 300 independent runs. Message senders are selected randomly across participants, and message addresses (grid squares) are randomly chosen by selecting a (different) node and addressing the message to its most frequented grid square as defined by its location profile. Our simulation was concerned with measuring the effectiveness of PPBR over metropolitan areas, and as such, we did not simulate local flooding. We considered a message successfully delivered if it reaches the destination address. The grid overlay consists of 200 m $\times$ 200 m grid squares, roughly the size of a city block, and we chose d — the requisite travel distance of a node before transitioning to the passing phase — to be the size of a grid square (200 m).

4.1 Simulation Settings and Inputs

Datasets. Due to privacy constraints, the number of realistic datasets that are suited for evaluation is unfortunately limited. We require that the data contain not only a large number of nodes, but also that the movement of the nodes should express regular routines over an extended collection time (*i.e.*, many days). There is considerable work in constructing models for human movement [1, 4, 16, 23, 25, 28]; however, most of these models do not realistically simulate movement over long periods, nor do they model regularity. There also exists extensive catalogs of real world movement traces, such as the CRAWDAD repository [27]; unfortunately, most of the traces are either too short with too few nodes or do not contain fine-grained location information.

To demonstrate the feasibility of our routing protocols, we utilize a suitable real-world data trace as well as a synthetic trace of human movement (summarized in Table 1):

- **Cabspotting:** The **Cabspotting Dataset** [34] contains GPS coordinates and timestamps of 536 taxicabs in the San Francisco area. The dataset spans 20 days: from May 20, 2008 until June 7, 2008. It should be noted that although the movements of taxis are not representative of the general population (taxis are arguably more mobile than the average person), simulations using this dataset can be interpreted as representing a network composed of the taxi drivers' smartphones.
- **SLAW:** We require a synthetic model that (i) accurately represents human *flight patterns*, (ii) contact rates, (iii) *waypoints* (popular places), and (iv) routines. The closest model to meeting our needs is **Self-similar Least Action Walk** (SLAW) [28]. Based in part on Levy walks [35], SLAW introduces a protocol called *Least Action Trip Planning* (LATP) that produces human-like trips between fractal waypoints, that are themselves determined by finding hotspots in actual GPS traces. Lee *et al.* showed that SLAW produces more human-like inter-contact times and flight paths than other leading movement models [16, 25, 29].

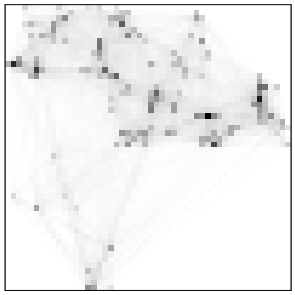

Figure 2: Heatmap of the General Node Profiles for the SLAW dataset. Darker shades indicate regions with higher node densities.

Node Contacts. For two nodes to make contact, they must be in the same location at the same time. However, the periodicity of location entries in the Cabspotting dataset is not consistent across nodes (or for the same node). We consider two nodes to have made contact if they are within 10 meters in a 10 second window. In SLAW, a location entry is generated every 60 seconds consistently across all nodes; we consider a contact to occur if two nodes are within 10 meters at the same minute mark.

Timeouts. We use a 12 hour local timeout in simulations with both traces. The global timeout varied between the two datasets. For the shorter, more dense SLAW movement trace, a three day global timeout is used. The longer, more sparse Cabspotting trace uses a seven day global timeout. Finally, simulations begin after an initial delay so that node profiles can be well seeded; delays of three and seven days are used for SLAW and Cabspotting, respectively.

Location Profiles. Each node constructs its location profile using a three day window of location histories. Location profiles are updated daily (*e.g.*, while the phone is charging), and the current day's profile represents the location history of the three previous days.

To generate the general node profile, we select a 10% sample of nodes from each dataset and use three days worth of movement data. The 10% sample is excluded from all simulation experiments. A visualization of the resulting general node profile for the SLAW dataset is shown in Figure 2.

4.2 Simulation Results

To measure the efficiency of PPBR, we compare our strategy against two probabilistic protocols that do not use location information: *probabilistic random walk* and *probabilistic flooding*. The probabilistic random walk routing scheme also has passing and holding phases; however, unlike PPBR, the random walk does not use location profiles. Instead, a node accepts a carrier's advertised message with a fixed probability of $1/k$ (*i.e.*, 10%). The random walk protocol allows us to measure both the effectiveness of using location information as well as the local threshold selection process.

Additionally, we compare PPBR to a 10% probabilistic flood in which nodes duplicate the message to a contacted node with probability 0.1. The flood provides insight into a worst case for network load – *i.e.*, exponential growth in the number of duplicate messages. The global and local timeouts for both random protocols are identical to those used by PPBR.

Threshold Estimation. As described in Section 3.2, each node computes its threshold marginal similarity score (τ) based on the general node profile and its knowledge of the routing area. Ideally, τ should be chosen such that a message is transferred to exactly one of the k nodes that a carrier encounters during its passing phase. To determine if our local, per-node threshold calculations were generating good thresholds, we looked at the variance of thresholds calculated at each node for one day in the simulation. Intuitively, a low variance indicates that nodes are independently able to reach a consensus as to a good value for τ, without exchanging any information amongst themselves. The average value for τ was 1.557 and 1.353 for SLAW and Cabspotting, respectively. We found that there is very low variance among the nodes' thresholds: 0.011 for SLAW and 0.085 for Cabspotting. Similar results were found for other days in the simulation, and as we show below, these thresholds result in both low latency and low network load.

Performance Metrics. We evaluate our routing performance using the following metrics: *delivery rate* is the percentage of messages that reach the destination address (a grid square); *latency* is the amount of time it takes for a message to be delivered; and *network load* is the number of messages in the network at a given time. Ideally, the routing protocol should deliver messages with a high delivery rate, low latency, and low network load.

Delivery Rate and Latency. Table 2 lists the delivery rates and latencies for PPBR, random walk, and probabilistic flooding. Unsurprisingly, flooding offers both the best latency and delivery rates. (As we show later, it also incurs a very high network load, making it impractical for networks of battery-constrained smartphone devices.) PPBR routing outperforms random walk for both median latency and delivery rate. Although the average latency for PPBR using the Cabspotting dataset is 0.8 hours slower, the median latency is nearly an hour faster. The skew in the average latency is caused in part by the higher delivery rate, and that some messages were delivered later in the simulation after random walk was no longer delivering messages.

It is worth emphasizing that the delivery rates reported in Table 2 result from single attempted transmissions. The sender can increase the delivery rate by sending redundant copies sufficiently spaced in time to allow different sets of carriers to deliver the message.

Network Load. The load on the network is measured as the average number of message duplicates in the system across all simulations runs. PPBR does not guarantee that only a single copy of a given message is present in the system. Carriers announce a message to k other nodes; ideally, only one node *should* accept it. If the message is accepted,

	Nodes	Length	Area	Contact Rate	Waypoints
SLAW [28]	1000	7 days	100 km^2	12.62 per hour	150
Cabspotting [34]	536	20 days	326 km^2	1.17 per hour	n/a

Table 1: Characteristics of the movement data sets.

	Cabspotting		**SLAW**	
	Med/Avg Latency (hrs)	Rate	Med/Avg Latency (hrs)	Rate
PPBR	3.6/6.8 [1.2,4.6]	62.6%	4.2/4.8 [2.6,6.2]	61.8%
Walk-10%	4.4/6.0 [1.6,8.1]	43.4%	5.1/5.5 [2.9,5.2]	48.0%
Flood-10%	2.8/4.1 [1.6,4.4]	99.4%	3.4/3.3 [2.2,4.2]	100.0%

Table 2: Median and Average Latencies (first and third quartiles in braces) and Delivery Rate.

the carrier retains the message until either it is delivered or a local timeout occurs. Hence, each message could potentially have multiple (or zero) duplicates in the system.

Figure 3 plots the number of messages that persist in the system over time, normalized to the number of senders in the system (which, in our simulation experiments is always 300). The average number of message copies, computed over the entire simulation, is shown in the Figure's key. Note that the number of message duplicates may be less than one if either some messages are not accepted by any of the k encountered nodes, or if all message copies are delivered to their destinations. As expected, flooding incurs significant network load, resulting in approximately two orders of magnitude more message copies than PPBR. Although the number of duplicates is slightly larger for PPBR than our naïve random walk protocol, the load is easily manageable.

Figure 3 further validates the effectiveness of the threshold values. For both datasets, the number of message copies remains relatively stable throughout the simulations. The probabilistic "best-of-k" scheme employed by PPBR incurs only small network loads, highlighting its scalability and practicality.

5. SECURITY PROPERTIES

This section analyzes the security guarantees of PPBR in the presence of adversaries who wish to perform PROFILING, DE-ANONYMIZATION, and DISRUPTION attacks.

Profiling. All message exchanges in PPBR occur in the open, and an adversary can observe any exchange in its presence. However, PPBR offers strong privacy protections against PROFILING attacks during a message exchange for both the node announcing a message as well as the node who receives, and possibly accepts, the message announcement.

Message Exchange Carrier Protections: An adversary can determine that a carrier node who advertises a message has a high marginal similarity to the message's address; otherwise, the node would not be advertising the message. More precisely, the adversary knows that the marginal similarity for the carrier is lower bounded by the threshold τ.

By design, nodes choose τ such that they should expect to accept messages addressed to $1/k$ of the grid squares. Hence, *the acceptance of a message does not necessarily indicate that the message's address is particularly important to the node that accepted it*. Depending upon the value of k, a node may be expected to accept messages targeted at hundreds of grid squares across the routing area. An adversary cannot conclude that a message was accepted because the message's address is frequently visited by the advertising node. Moreover, as we show below, a node may not even accept a message addressed to a grid square for which it is very familiar.

The choice of k has privacy and performance implications, and a clear tradeoff exists: Larger values of k decrease privacy since nodes accept messages for fewer locations, and thus an adversary could deduce that these locations are more likely relevant to the victim node. Conversely, smaller values of k increase privacy since nodes accept messages to more locations, further obscuring which are important. Smaller values of k also incur higher power consumption and network load as more nodes will likely accept (and transfer) the message. In our simulation studies, we found that $k = 10$ achieves reasonable privacy while restraining the number of message transfers, and we use this value for the experiments described below.

To study this tradeoff further, we determined for each node the set of addresses (grid squares) that would result in its acceptance of a message. We then compared this set of addresses to the nodes' most frequented locations as defined in their location profiles. As expected, nodes accepted messages addressed to $1/k$ of the grid squares, on average. However, many of those locations correspond to grid squares that would be uninteresting to an adversary concerned with PROFILING. If we consider an adversary who is interested in the most frequented grid squares of a victim node – that is, the highest value grid squares in the node's location profile – these "interesting" grid squares comprise only a small fraction of the total locations for which a node would accept a message.

This relationship is depicted in Figure 4 (left). The curves represent the averages across all nodes in the Cabspotting and SLAW datasets. The x-axis denotes the number of points an adversary is interested in (*i.e.*, the x grid squares most frequented by the node). The y-axis plots the fraction of the locations that are accepted by the node which are of interest to the adversary. For example, using the Cabspotting dataset, 38% of announced messages belong to the advertising node's 800 most frequented locations. If the adversary is interested in a node's 200 most frequented grid squares, just 10% of advertised messages belong to this interest set. More generally, the more specific the adversary's interest, the more difficult it is for him to distinguish the pertinent message addresses that are announced by a node, and consequently, the more difficult it is to discover the node's most frequented locations.

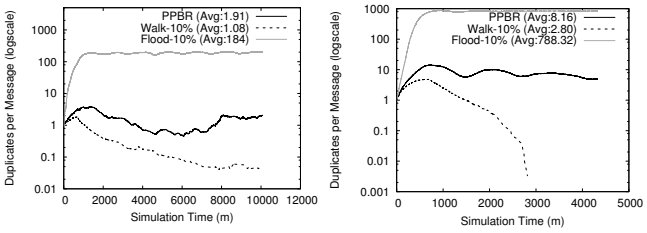

Figure 3: The number of message copies ("duplicates") of each message for (**left**) Cabspotting and (**right**) SLAW, and inset, the average.

The adversary's ability to discern profile information is further diminished due to our algorithm's willingness to discard announcements that are targeted at highly frequented areas. That is, a significant portion of the grid squares that are most frequented by a node actually have low marginal similarity. Recall that the marginal similarity is the ratio of the node's similarity score to the general node profile's similarity score. Hence, if a message is addressed to a grid square that is often frequented by the node *but also highly frequented according to the general node profile*, then the ratio will not exceed the τ threshold, and the node will *never* accept a message addressed there. Consequently, such interesting locations are unobservable and *safe* from adversarial analysis.

Figure 4 (right) visualizes this relationship. Again, the x-axis considers the number of grid squares an adversary would find interesting for a victim node. The y-axis represents the fraction of those interesting grid squares a node would *never* accept a message for, averaged across all nodes. For example, consider an adversary interested in the top 200 most frequent locations of a node: In the Cabspotting data set, 68% of those locations are safe from analysis by an adversary.

Message Exchange Receiver Protections: During the passing phase, receivers do not acknowledge acceptance (or rejection) of a message, and hence an adversary cannot directly determine its similarity to the message's destination address.

An adversary who is able to follow the node for a distance of at least d can determine whether the message has been accepted by observing whether or not it is re-advertised by the node. However, since the node is physically followed, such a stalking attack inherently leaks the victim's location information regardless of the particular routing protocol being used (and hence, as described in Section 2, stalking attacks are outside of our threat model). Regardless, if the node *is* followed, or if a separate colluding eavesdropper discovers that the node later advertised the message, then the adversary can conclude that the node accepted the message. In such cases, the effectiveness of a PROFILING attack against the receiver is identical to the effectiveness against a carrier advertising a message (see above).

De-Anonymization. The standard addressing primitive of HumaNets is geocast, and thus all participants at the addressed location at the time of delivery should receive the message. Receiver anonymity is not protected in HumaNets because an adversary located in the address location trivially learns the identities of the message recipients by simply observing them.

However, PPBR provides in-transit anonymity for message originators (or senders). An intercepted message, past the initial hop, cannot be traced to the original sender without completely retracing the message's path. If an adversary is witness to the initial hop of a message, the originating sender may be exposed. We note, however, that this is similar to the level of protection provided by many Internet-based anonymity systems (*e.g.*, Tor [12]) in which an adversary on the first hop trivially learns the message's sender.

It is worth noting that message replay attacks in which an attacker re-injects a message in hopes of discovering its path are also infeasible. It is highly unlikely a message will take the same path due to variability in human movement.

Disruption. PPBR also provides protection against DISRUPTION attacks in which an adversary attempts to intercept messages in the network. If the attacker is able to infiltrate the network and receive a large portion of the k handoffs for each message, then the probability that the message will be transferred to an honest node is reduced. However, such an attack may also be prohibitively expensive for an adversary since message exchanges occur whenever two participants have a chance encounter. Additionally, such an attack may be mitigated by adjusting the number of passing attempts (*i.e.*, k) to compensate for the attacker's presence.

PPBR's SCALABILITY property also makes it resistant to denial-of-service attacks in which the attacker attempts to overwhelm the network's resources by injecting spurious messages. Although an attacker may inject wasteful messages into the HumaNet, the impact of each additional message on the network is linear, by design. In comparison, each additional message in a flooding protocol incurs an exponential increase in load on the network, and a few injected messages may be sufficient to overload the network.

6. RELATED WORK

Location-Based Routing. The ability to leverage geographic information to efficiently route packets has been well explored in the literature. In many instances, these techniques require participants to announce their locations. For example, Last Encounter Routing (LER) [18] and ProPHET [30] expose location information; LER assumes that the network is sufficiently connected to allow stable and longstanding paths. The Bubble protocol [21] uses social networks to efficiently route messages, but allows any party to discover social relationships. Although these techniques may efficiently route messages, they are not well-suited for settings in which the disclosure of location histories and/or social relationships may be cause for government-imposed punishment. We desire protocols that efficiently and scal-

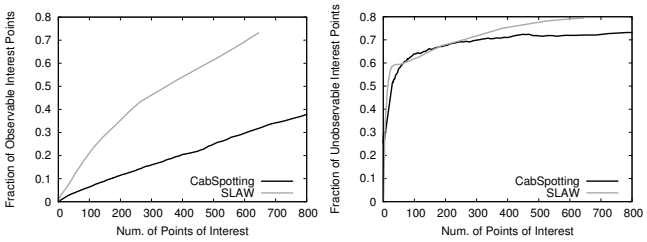

Figure 4: Fraction of Safe Interest Points (**left**) and Fraction of Interesting Observations (**right**).

ably deliver messages while preserving users' location histories and social relationships.

Location-based routing has also been studied in the context of wearable computing. Of particular relevance is Davis *et al.*'s geographic-based routing protocol [10]. There, the authors use flooding techniques to disseminate messages when the network's devices are storage constrained; they consider a pruning approach in which nodes drop messages that are addressed to locations that they have not recently visited. Our routing techniques rely on similar heuristics, but take a more proactive approach by targeting potential message carriers who are *likely* to visit a message's destination. Similarly, *pocket-switched networks* [7, 8, 20] provide methods of routing messages between pocket-sized devices. However, the protocols are intended for small area routing (*i.e.*, at the scale of an academic conference) and focus on reliability. Our protocols are designed specifically for smartphones, leverage the devices' ubiquity and location-awareness, and target city-scale routing.

Location Privacy. There are a number of approaches that attempt to preserve *location privacy*. Here, the goal is often to prevent an adversary from either identifying the source of an intercepted communication or tracking a node over time.

Several protocols [15, 26, 37, 45] achieve location privacy by relying on ephemeral pseudoidentities. Such approaches provide *unlinkability* by impeding an adversary's ability to associate different broadcasts with the same node. Although these techniques can be used in conjunction with our PPBR protocol, we assume an adversary who is physically present at various (but not all) locations in the network and can identify individuals and associate broadcasts with their senders (*e.g.*, through physical identification and message triangulation). Similarly, anti-localization techniques [31] that are designed to prevent an adversary from determining a sender's location [22] are ineffective in our context in which the adversary physically observes nodes.

A number of location privacy protocols are loosely based off of AODV [33], a popular routing protocol for decentralized mobile networks (*e.g.*, MANETs). However, such techniques assume a highly connected and mostly static network in which messages can be quickly forwarded between nodes. For example, the ALARM [11] routing system privately disseminates topology snapshots to participating nodes, AO2P [42] assumes mostly static positions and immediate connectivity between nodes, PRISM [13] assumes a trusted third party and longstanding paths that can be used to route traffic, and ODAR [39] relies on source routing. Similarly, the ANODR [26] system and its extensions [37, 43] enable anonymous communication in a MANET by establishing onion-like structures [40] that obscure the identity of the sender. SDAR [6] also uses onion-like routing, but uses a "trust management system" in which nodes choose which peers to route messages towards based on their level of trust of those nodes.

These protocols assume that nodes are mostly stationary, communication can occur with low latency, and anonymous paths can be reused for multiple exchanges. They are not well-suited for networks of mobile smartphones where immediate connectivity is not available, nodes are highly mobile, and paths cannot be predicted *a priori*. In contrast, we desire protocols that take advantage of routine movement and do not require human operators to change their habits in order to participate, *even if such a requirement limits opportunities for exchanging messages*. Our setting therefore requires *delay tolerant networks* (DTNs) in which messages are stored and forwarded only during chance encounters.

There are a number of existing DTN protocols that are similar to HumaNets, but either have limited functionality or lack HumaNets' privacy protections. For instance, Zebranet [24] uses local information to efficiently exchange information between sensor nodes in order to track wildlife. However, the network can route messages only towards fixed basestations. GeoDTN+Nav [9] is a vehicular ad-hoc network routing scheme that, like HumaNets, relies on location profiles to deliver messages in a DTN. However, GeoDTN+Nav requires that at least some nodes follow fixed paths (*e.g.*, bus routes) or provide their destinations before travel (*e.g.*, via a car navigation system).

The work that perhaps most closely resembles ours is Shifka *et al.*'s protocol [38]. Here, the authors use the heuristic that nodes that share more *contexts* are more likely to encounter one another. Like our approach, participants construct profiles that describe frequented locations. To provide profile confidentiality, their technique relies on public encryption with keyword search (PEKS) to limit the adversary's ability to enumerate the contents of a profile. Additionally, their approach assumes a trusted third party (TTP) that assigns attribute values (*e.g.*, a frequented location) to nodes. In contrast, HumaNets does not require a TTP, and allows nodes to self-determine their profiles.

Extending HumaNets. Aviv *et al.* introduced the *polygon-intersection protocol* for HumaNet routing [3]. Their technique requires nodes to announce their frequented locations (*i.e.*, the areas they travel), and consequently leaks significant information about the network's participants. This paper builds off their work by refining the threat model and introducing novel *privacy-preserving* decentralized routing techniques that minimize the exposure of information.

7. CONCLUSION

This paper presents *probabilistic profile based routing* (PPBR), a novel privacy-preserving geographic messaging protocol for HumaNets. Designed for networks of smartphone devices, our PPBR routing protocol avoids the use of the cellular network — or any other centralized infrastructure — and is well-suited for environments in which traditional communication is subject to monitoring and/or censorship. PPBR leverages self-determined location profiles to assist routing while minimizing the disclosure of location information to outside observers as well as adversaries who infiltrate the network. In particular, we demonstrate that PPBR is resistant to disruption, de-anonymization, and location-leakage attacks.

Using simulations over real-world and synthetic movement data, we show that PPBR provides reasonable delivery rates and latency. Unlike flooding approaches, our probabilistic routing algorithm does not require exponential message transfers, and is therefore appropriate for networks of battery-constrained smartphones. Our future work includes adapting PPBR to provide long-distance (state- and country-scale) messaging.

References

[1] Udel models, July, 2010. `http://www.udelmodels.eecis.udel.edu/`.

[2] 3rd Generation Partnership Project. Universal Mobile Telecommunications System (UMTS); Synchronization in (UTRAN) Stage 2. Technical Specification Group Services and System Aspects 3GPP TS25.402 v8.1.0, 3rd Generation Partnership Project, July 2009.

[3] A. J. Aviv, M. Sherr, M. Blaze, and J. M. Smith. Evading Cellular Data Monitoring with Human Movement Networks. In *USENIX Workshop on Hot Topics in Security (HotSec)*, August 2010.

[4] F. Bai, N. Sadagopan, and A. Helmy. IMPORTANT: A Framework to Systematically Analyze the Impact of Mobility on Performance of Routing Protocols for Adhoc Networks. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2003.

[5] N. Bilton. Tracking File Found in iPhones. The New York Times, April 20 2011.

[6] A. Boukerche, K. El-Khatib, L. Xu, and L. Korba. An Efficient Secure Distributed Anonymous Routing Protocol for Mobile and Wireless Ad Hoc Networks. *Computer Communications*, 28(10):1193–1203, 2005.

[7] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Pocket Switched Networks: Real-world Mobility and its Consequence for Opportunistic Forwarding. Technical Report 617, University of Cambridge, Febuary 2005.

[8] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Impact of Human Mobility on Opportunistic Forwarding Algorithms. *IEEE Transactions on Mobile Computing*, 6(6):606–620, 2007.

[9] P. Cheng, J. Weng, L. Tung, K. Lee, M. Gerla, and J. Haerri. GeoDTN+Nav: A Hybrid Geographic and Dtn Routing with Navigation Assistance in Urban Vehicular Networks. In *Symposium on Vehicular Computing Systems*, 2008.

[10] J. A. Davis, A. H. Fagg, and B. N. Levine. Wearable Computers as Packet Transport Mechanisms in Highly-Partitioned Ad-Hoc Networks. In *IEEE International Symposium on Wearable Computers*, 2001.

[11] K. E. Defrawy and G. Tsudik. ALARM: Anonymous Location-Aided Routing in Suspicious MANETs. *IEEE Transactions on Mobile Computing*, 10:1345–1358, 2011.

[12] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX)*, 2004.

[13] K. El Defrawy and G. Tsudik. PRISM: Privacy-friendly Routing in Suspicious MANETs (and VANETs). In *International Conference on Network Protocols (ICNP)*, 2008.

[14] N. Fathi. Iran Disrupts Internet Service Ahead of Protests. The New York Times, February 10 2010.

[15] J. Freudiger, M. H. Manshaei, J.-P. Hubaux, and D. C. Parkes. On Non-cooperative Location Privacy: A Game-theoretic Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[16] J. Ghosh, S. J. Philip, and C. Qiao. Sociological Orbit Aware Location Approximation and Routing (SOLAR) in MANET. *Ad Hoc Networks*, 5(2):189–209, 2007.

[17] D. Gonzales and S. Harting. Can You Hear Libya Now? The New York Times, March 4 2011.

[18] M. Grossglauser and M. Vetterli. Locating mobile nodes with ease: learning efficient routes from encounter histories alone. *IEEE/ACM Trans. Netw.*, 14(3):457–469, 2006.

[19] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[20] P. Hui, A. Chaintreau, J. Scott, R. Gass, J. Crowcroft, and C. Diot. Pocket Switched Networks and Human Mobility in Conference Environments. In *ACM SIGCOMM Workshop on Delay-tolerant networking (WDTN)*, 2005.

[21] P. Hui, J. Crowcroft, and E. Yoneki. BUBBLE Rap: Social-Based Forwarding in Delay Tolerant Networks. *IEEE Transactions on Mobile Computing*, 99, 2010.

[22] N. Husted and S. Myers. Mobile Location Tracking in Metro Areas: Malnets and Others. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[23] A. Jardosh, E. M. Belding-Royer, K. C. Almeroth, and S. Suri. Towards Realistic Mobility Models for Mobile Ad Hoc Networks. In *International Conference on Mobile Computing and Networking (MOBICOM)*, 2003.

[24] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002.

[25] M. Kim, D. Kotz, and S. Kim. Extracting a Mobility Model from Real User Traces. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2006.

[26] J. Kong and X. Hong. Anodr: Anonymous on demand routing with untraceable routes for mobile ad-hoc networks. In *ACM International Symposium on Mobile Ad Hoc Networking and Computing*, 2003.

[27] D. Kotz and T. Henderson. CRAWDAD: A community resource for archiving wireless data at dartmouth. `http://crawdad.cs.dartmouth.edu/`.

[28] K. Lee, S. Hong, S. J. Kim, I. Rhee, and S. Chong. SLAW: A New Mobility Model for Human Walks. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2009.

[29] S. Lim, C. Yu, and C. Das. Clustered Mobility Model for Scale-Free Wireless Networks. In *IEEE Conference on Local Computer Networks (LCN)*, 2006.

[30] A. Lindgren, A. Doria, and O. SchelÃl'n. Probabilistic routing in intermittently connected networks. In P. Dini, P. Lorenz, and J. de Souza, editors, *Service Assurance with Partial and Intermittent Resources*, volume 3126 of *Lecture Notes in Computer Science*, pages 239–254. Springer Berlin / Heidelberg, 2004.

[31] X. Lu, P. Hui, D. Towsley, J. Pu, and Z. Xiong. Anti-localization Anonymous Routing for Delay Tolerant Network. *Computer Networks*, 54(11):1899 – 1910, 2010.

[32] P. Mann. Timing Synchronization for 3G Wireless. *EE Times Asia*, December 2004.

[33] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, IETF, 2003.

[34] M. Piorkowski, N. Sarafijanovoc-Djukic, and M. Grossglauser. A Parsimonious Model of Mobile Partitioned Networks with Clustering. In *Conference on COMmunication Systems and NETworkS (COMSNETS)*, 2009.

[35] I. Rhee, M. Shin, S. Hong, K. Lee, and S. Chong. On the Levy-Walk Nature of Human Mobility. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2008.

[36] M. Richtel. Egypt Cuts Off Most Internet and Cell Service. The New York Times, January 28 2011.

[37] S. Seys and B. Preneel. ARM: Anonymous Routing Protocol for Mobile Ad hoc Networks. In *International Conference on Advanced Information Networking and Applications (AINA)*, 2006.

[38] A. Shikfa, M. Onen, and R. Molva. Privacy and Confidentiality in Context-Based and Epidemic Forwarding. *Computer Communications*, 33(13):1493–1504, 2010.

[39] D. Sy, R. Chen, and L. Bao. ODAR: On-Demand Anonymous Routing in Ad Hoc Networks. In *IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*, 2006.

[40] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous Connections and Onion Routing. In *IEEE Symposium on Security and Privacy (Oakland)*, 1997.

[41] M. Wegener. Operational urban models state of the art. *Journal of the American Planning Association*, 60(1):17–29, 1994.

[42] X. Wu and B. Bhargava. AO2P: Ad Hoc On-Demand Position-Based Private Routing Protocol. *IEEE Transactions on Mobile Computing*, 4:335–348, 2005.

[43] L. Yang, M. Jakobsson, and S. Wetzel. Discount Anonymous On Demand Routing for Mobile Ad hoc Networks. In *ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2006.

[44] A. C. Yao. Protocols for Secure Computations. In *Symposium on Foundations of Computer Science (FOCS)*, 1982.

[45] Y. Zhang, W. Liu, W. Lou, and Y. Fang. MASK: Anonymous On-Demand Routing in Mobile Ad Hoc Networks. *IEEE Transactions on Wireless Communications*, 5(9):2376–2385, September 2006.

CIS 380 Project 2
PennOS: UNIX-like Operating System Simulation

"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity."
Dennis Ritchie

Adam Aviv*
with Vin Mannino, Thanat Owlarn, Seth Shannin, and Kevin Xu

MILESTONE: Nov. 9-11
DUE: Nov. 30 @ 10pm
Demos: Dec. 1-2

Directions
This is a group project. You must work in groups of four. You may reuse code from previous projects, **but only code you wrote**.

Overview
In this assignment you will implement *PennOS*, your own UNIX-like operating system. PennOS is designed around subsystems that model those of standard UNIX. This will include programming a basic priority scheduler, flat file system, and user shell interactions. Unlike a *real* operating system, you will not be required to actually boot on hardware; rather, your PennOS will run as a guest OS within a single process running on a host OS.

How to use this document

The following specification provides a road map for completing this project; however, as you develop your code, you may find it necessary to deviate from the specification. **In particular, you will likely find it useful/necessary to change the type definitions of some of the functions to better match your development, or provide additional shell/user level functions to support debugging.** Not only is this encouraged, but it is expected. If you are ever in doubt about a design decision, ask your friendly TAs, and be sure to document such changes in your `README` and companion document.

*Based on previous projects written by Sandy Clark & Micah Sherr, who based their version on a more previous versions by Stefan Miltchev, Eric Cronin, Guarav Shah, Hee Hwan Kwak, Stuart Eichert, Scott Raven, Jon Kaplan, Robert Spier, & Dianna Xu

1 Specification

There are two states in which an operating system exists: (1) *kernel land* and (2) *user land*. During execution, an operating system switches between these two states continuously. As an example, consider what happens when a program issues a system call. First, the system call is executed in user land which hits a hook; that is, the running process actually calls the system call. Next, the operating system must handle the system call, switching from user to kernel land. Once the system call completes, the operating system returns control to the calling process, returning back to user land.

In the previous projects, you have interacted with the operating system at the user land level, and in this project, you will take a peak behind the curtain at kernel land. Well, not exactly, but you will simulate the basic functionality of an operating system by programming your own operating system simulator *PennOS*. Using the `ucontext` library, you will implement a basic priority scheduler; additionally, you will implement a flat file system for your operating system to mount, and a basic shell and programming API for a user to interact with your operating system.

Unlike a *real* operating system, your PennOS is not required to boot on hardware (or handle devices in general); instead, it will run as a single process on a host OS. The `ucontext` library is similar to a threading API in that allows one process to split its resources across multiple instances. `ucontexts` do not provide a scheduler like you may be used to with traditional threading, and your first task will be implementing a `SIGALRM`-based scheduler for context switching.

Another critical part of an operating system is handling file reads and writes. Your operating system will mount a single file system, FlatFAT: a simple file system implementation based on FAT. Your FlatFAT implementation will be stored within a single file on the host file system, and will be mounted by PennOS in a *loopback* like manner. Additionally, unlike traditional file systems, FlatFAT is only required to handle files within a single top level directory. You are required to allow the creation, modification, and removal of files under the top level directory.

The last part of your operating system is providing user land interaction via a simple shell. You will program this shell using the user land system calls providing by PennOS. Your shell will provide job control, `stdin`/`stdout` redirection, and a functional set of built-in commands for testing and exploring your operating system.

One last note: **This is a long and complex project that will take you many, many hours to complete.** This document can only provide you with a primer of the innumerable challenges you will face; read it carefully, but be sure to use other resources as well. In particular, read the manual pages, and ask questions of your friendly TAs. Likely, this will be the largest program you will write as an undergraduate student, so divide and conquer and plan ahead so that the pieces fit together. Remember, a lot of small, easy programs equal one large, complex program.

1.1 Function Terminology

Your operating system will provide a number of different functions and interfaces. The following symbols indicate how the functions are to be used:

- **(K)**: indicates a kernel level function that may only be called from the kernel side of the operating system.
- **(U)**: indicates a user level function that may only be called from the user side of the operating system. These are your operating system's system calls.
- **(S)**: indicates a built in program that can be called directly from the shell.

Keep in mind that your implementations need not match the function definitions in this document exactly. Although you are required to meet the core functionality described herein, the function definitions provided are suggestions: **You may find it useful/necessary to pass extra parameters to some of the functions we describe. You may also find it useful to add additional kernel/system/user level functions as you see fit as long as they are in the spirit of the assignment.**

1.2 PennOS Processes/Threads

Your PennOS operating system will run as a single process on the host OS (e.g., the *speclab* machine). That is, each of the "processes" in PennOS is really the same process as PennOS according to the host operating system, but within PennOS, each process will be separated into context threads that are independently scheduled by PennOS. To accomplish this task you will be using the `ucontext` library: **If you are issuing calls to `fork(2)`, `exec(2)`, or `wait(2)`, you are doing something very wrong**. Despite being context threads[1], your operating system will treat them like processes and will organizing them into **p**rocess **c**ontrol **b**locks (or *PCB*'s).

A PCB is a structure that describes all the needed information about a running process (or thread). One of the entries will clearly be the `ucontext` information, but additionally, you will need to store the thread's process id, parent process id, children process ids, open file descriptors, priority level, etc. Refer to Steven's *Advanced Programming in the UNIX Environment* and Tanenbaum's *Modern Operating Systems* for more information about data normally stored in a PCB. **You must describe your PCB structure in your** `README`.

1.2.1 Process Related Required Functions

Your operating system must provide the following user level functions for interacting with PennOS process creation:

- `p_spawn(void * func)` (U) will fork a new thread that retains most of the attributes of the parent thread (see `k_process_create`). Once the thread is spawned, it will execute the function referenced by `func`.
- `p_kill(int pid, int signal)` (U) kill the thread referenced by pid with the signal `signal`.
- `p_wait(int mode)` (U) set the calling thread as blocked (and do not return) until a child of the calling thread changes state. `p_wait()` returns a structure with two fields: `int pid` - the process id of the child thread that changed state, and `int status` - indicating the state of the child. The `mode` argument should be used to indicate a `NOHANG` condition. In this case, `p_wait` should not block and should return `NULL` immediately if there are no child threads to wait on. If the calling thread has no children, `p_wait` should return `NULL` immediately.
- `p_exit()` (U) exit the current thread unconditionally.

Additionally, your operating system will have the following kernel level functions:

- `k_process_create(pcb_t * parent)` (K) create a new child thread and associated PCB. The new thread should retain much of the properties of the parent. The function should return a reference to the new PCB.

[1] We use "context" and "thread" interchangeably in this document to describe a PennOS process.

- `k_process_kill(pcb_t * process, int signal)` (K) kill the process referenced by `process` with the signal `signal`.
- `k_process_terminate(pcb_t * process)` (K) called when a thread returns or is terminated. This will perform any necessary clean up, such as, but not limited to: freeing memory, setting the status of the child, *etc.*

1.2.2 Zombies and Waiting

As processes complete, it may not be the case that their parent threads can wait on them immediately. If that is the case, you must queue up these threads so that the parent may wait on them in the future. These threads are *Zombies*[2]. You will likely have a zombie queue for each thread, referenced in the PCB. If at any time the parent thread exits without waiting on zombie children, the zombie children should immediately die, as well as non-zombie children threads. Note, this is similar to how `INIT` inherits orphan child processes and kills them off.

Additionally, as noted above, other child process state changes can cause a `p_wait()` to return. In UNIX, a child process that transitions from running to stopped would issue a `SIGCHLD` signal. Your operating system also should have functionality for parent process to learn of similar state changes. In your PennOS kernel land implementation of `p_wait()` you will find it useful to maintain other queues of "waitable" children, not just zombie children.

1.2.3 Signals in PennOS

Your operating system will not have traditional signals and signal handlers. (However, the host operating system may deliver signals that you must handle.) Instead, signaling a PennOS process indicates to PennOS that it should take some action related to the signaled thread, such as change the state of a thread to stopped or running. Your operating system will minimally define the following signals:

- `S_SIGSTOP`: a thread receiving this signal should be stopped
- `S_SIGCONT`: a thread receiving this signal should be continued
- `S_SIGTERM`: a thread receiving this signal should be terminated

If you would like to add additional signals, be sure to document them and their functionality in your `README` file.

1.2.4 Process Statuses

PennOS will provide *at least* the following user level functions/macros that will return booleans based on the status returned from `p_wait`.

- `W_WIFEXITED(status)`: return true if the child terminated normally, that is, by a call to `p_exit` or by returning.
- `W_WIFSTOPPED(status)`: return true if the child was stopped by a signal.
- `W_WIFCONTINUED(status)`: return true if the child was continued by a signal.

[2] Mmmm ... brains.

- `W_WIFSIGNALED(status)`: return true if the child was terminated by a signal, that is, by a call to `p_kill` with the `S_SIGTERM` signal.

1.3 Programming with User Contexts

`ucontext` is a basic thread-like library provided on most UNIX systems. Essentially, it allows a user to isolate some portion of code execution within a *context* and switch between them. On the course website, we have provided a sample program that demonstrates how to switch between contexts in a round robin fashion. A high-level description of context creation and execution is provided below, and more information can be found in the manual.

First, a `ucontext` structure must be initialized with a call to `getcontext(2)`. The structure will have the following fields (and more not shown):

```
typedef struct ucontext {
  struct ucontext *uc_link;
  sigset_t         uc_sigmask;
  stack_t          uc_stack;
  ...
} ucontext_t;
```

You still need to set the structure values above: `uc_link` is a the next context to run when this context completes[3]; `uc_sigmask` is a signal mask for blocking signals in this context; and, `uc_stack` is the execution stack used by this context. For a description of the `uc_stack` structure reference the manual for `sigaltstack(2)`. Setting these values is still insufficient to executed the context, and you still need to set up the function that will be called when the context is set or swapped. This done by a call to `makecontext(2)`, and it is well described in the manual. A context is switched in using either `setcontext(2)` or `swapcontext(2)`, which either directly sets the context, or sets and also saves the state of the running context, respectively.

We are leaving much of the details for you to learn on your own, but a good place to start is with a *Hello World* program for `ucontext`. We have provided one in the appendix (see Section **??**). Try editing that programming and adding new features. Here are some mini-exercise you might want to try: What happens if you want to switch back to the main program after printing "Hello World"? Can you write a program that alternates between two functions indefinitely? What happens when a signal is delivered? How do signals affect the execution of a context? How do you track the current context?

1.4 Priority Scheduler

Perhaps the most critical part of any operating system is the scheduler. This is the part of your operating system that chooses which program to run next. As described in class, most operating systems, including Linux, use a priority queue based scheduler. In your PennOS, you will also implement a priority scheduler, although it will be a much more simplified version with a fixed quanta.

1.4.1 Clock Ticks

You will schedule a `SIGALRM` signal to be delivered to your operating system every 100 milliseconds. We refer to this event as a *clock tick*, and on every clock tick the operating system will switch in and execute

[3] What might you want to set `uc_link` to?

the scheduler, which will then choose which process to run next. This may be any process that is *runnable* (i.e., not zombied, blocked, nor stopped) to execute, including the shell. To set an alarm timer at millisecond granularity, refer to `setitimer(2)`. Note that since your operating system is relying on `SIGALRM`, **non-kernel functions may not block** `SIGALRM`.

1.4.2 Priority Queues

Your operating system will have three priority levels: -1, 0, 1. As in standard UNIX, the lower a priority level, the more heavily the "process" should be scheduled. Child threads created by `k_process_create` should have a default priority level of 0, unless otherwise specified. For example, the shell should execute with priority level -1 because it is interactive.

Your priority queues will be relative. Threads scheduled with level -1 should run 1.5 times more often as threads scheduled with priority level 0, which run 1.5 times more often as threads scheduled with priority level 1. Within each priority queue, the threads are selected in a round robin format, and **each queue must be implemented as a linked list**. You may reuse your linked list implementation from previous projects. **You must also ensure that no thread is starved**.

As an example, consider the scheduling of the following threads with these priority levels: (1) `shell`, priority level -1; (2) `sleep`, priority level 0; and (3), `cat`, priority level 0. With a 100 millisecond quanta, after 10 seconds, what is the proportion of quanta for each process? First, there are 10 quanta per second, which means a total of 1000 quanta in 10 seconds. Of the available quanta, 600 quanta will be used for priority level -1, (that is the `shell`), since it must be scheduled 1.5 times as often. Of the remaining 400 quanta, 200 quanta will be used for `sleep`, and 200 quanta for `cat`.

To verify the correctness of your scheduler, you will implement a detailed logging facility that will generate a log entry for every clock tick. The specification of the log format is described in Section **??**; however, we will provide you with a tool to parse and analyze your log. The same tool will be used in grading your shell.

1.5 Running vs. Stopped vs. Blocked vs. Zombied

Threads can exists in four states: running, stopped, blocked, or zombied. A running thread may be scheduled; however, a stopped, blocked, or zombied thread should not be. A thread should only become stopped if it was appropriately signaled via `p_kill`. A thread should only be blocked if it made a call to `p_wait` or `p_sleep` (see below).

1.5.1 Required Scheduling Functions

Your operating system will provide *at least* the following user level functions for interacting with the scheduler:

- `p_nice(int pid, int priority)` (U) set the priority level of the thread `pid` to `priority`.
- `p_info(int pid)` (U) return a structure representing standard information about a thread `pid`. The structure must contain the following fields: `int status` : indicating the status of the thread; `char * command` : the name of the command executed by the thread (e.g., `cat`); `int priority` : indicating the priority level of the thread `pid`. You may supply additional fields, but be sure to document the structure in your `README`.

- `p_sleep(int ticks)` (U) set the thread `pid` to blocked until `ticks` of the clock occur, and then set the thread to running. Importantly, `p_sleep` should not return until the thread resumes running; however, it can be interrupted by a `S_SIGTERM` signal.

1.6 Flat File System

Your operating system will mount its own file system based on a FAT (file allocation table) file system, FlatFAT. Unlike traditional operating systems, all files will be allocated under a single, top level file directory, which greatly simplifies things. (For references, see Chapter 4.3 in Tanenbaum 3rd edition, pages 277 and 278.)

1.6.1 FATs and File System Blocks

Conceptually, you can think of a file as a bunch of fixed-size memory blocks, and a file system as a way to find the right blocks for a file in the right order. A FAT (or File Address Table) provides a simple way to do this. Placed at the beginning of a file system, the FAT is a fixed size and can be easily mapped into memory. This is also its greatest disadvantage: being of fixed size, the FAT also limits the size of the file system.

Your FAT will contain 512 entries, each row consisting of a short (1 byte), thus resulting in a table that is 1024 bytes wide. Each file block will be 1KB (1024 bytes) in size[4]. In a FAT, each table entry is a reference to the next table entry in the FAT. For example:

```
Physical | Link
---------+-------
    0    |  -1    <--- directory block
---------+-------
    1    |   4    <--- start of File A
---------+-------
    2    |   3    <--- start of File B
---------+-------
    3    |  -1    <--- last block of File B
---------+-------
    4    |   5    <--- second block of File A
---------+-------
    5    |  -1    <--- last block in File A
---------+-------
   ...   |  ...
```

The first block in a FlatFAT will always reference the directory file. The directory file is like any other file, but it stores a linked list encoding of all the files in the file system. The structure of the encoding is as follows:

```
struct node {
  struct node * next; //next file in file system
  char[256] fname; //name of the file, limited in size
  unsigned int size; //size of the file
  short fstart; //FAT entry for this block (1 byte wide)
}
```

[4] How large can the file system be?

By iterating over the directory file, all files in the file system can be enumerated. To find the block in the file system using a FAT entry, you will use `fseek(3)`. For example, given a FAT entry for the value 5, you can find that block in the file containing your FlatFAT by the following call to `fseek`:

```
fseek(flatfat_file, FAT_SIZE+5*BLOCK_SIZE, SEEK_SET)
```

where `FAT_SIZE` is the size of the FAT table, and `BLOCK_SIZE` is the size of a file block.

1.6.2 Formatting FlatFAT

Your file system will exist as a single flat file on the host OS, which must first be formatted. You will provide a separate program called `mkFlatFAT` that will properly format a file as a `FlatFAT`. Additionally, we will provide two programs that should be capable of parsing your formatted file system, which you are free to use to test your implementation. The first program is called `lsFlatFAT`, which will list all the files on a FlatFAT, similar to calling `ls -l`. The second program is called `catFlatFAT`, which will act like the `cat` program, except it will act on a FlatFAT file system.

1.6.3 Loading/Mounting FlatFAT

When loading/mounting your FlatFAT you will likely find it useful to `mmap(2)` the FAT portion of the file directly into memory. This way you will have copy-on-write for free. Here is a code snippet (minus error checking) to get you started on this process:

```
#define 1024 FAT_WIDTH

(. . .)

typedef short FAT_t;

int fat_fd;
FAT_t * fat;

fat_fd = open(flatfat_fs,O_RDWR);
mmap(fat_t, FAT_WIDTH, PROT_READ | PROT_WRITE, MAP_SHARED, fat_fd, 0);
```

Now, `fat` references an array, and you can proceed by loading the root directory via a function call such as `load_root(fat[0])`.

1.6.4 File System and Operating System Interaction

The role of the operating system is to protect the file system from corruption as well as manipulate it by reading, writing, and deleting files. Internally, the operating system will store a reference to a file descriptor (an integer) for each open file, as well as a structure indicating the mode for the file and relevant file pointers indicating where subsequent reads and writes should take place. The user side will reference a file by its file descriptor and manipulate the file via the user level interface described below.

Note that the file system-related system calls are abstraction layers and should not care about the format of the underlying file system. That is, consider what must occur when an operating system has mounted multiple file systems of different types. When there is a call to `read(2)` for a particular file descriptor, the operating system will determine on which file system variant the file lives (e.g., `FAT32`, `ext3`, etc.)

and then call the appropriate file system dependent function to perform the read operation (which usually is provided by module).

PennOS must work in a similar way, particularly when considering how to handle the `stdin` and `stdout` file descriptors. Essentially, PennOS must manipulate two types of file descriptors, those for `FlatFAT` and those for `stdin` and `stdout`. A user level program should be able to write to `stdout` using the same interface as it would write to a `FlatFAT` file. **If a user level program is calling `read(2)`, then you are doing something wrong.**

1.6.5 Required File System Related System Calls

Your operating system will provide *at least* the following functions for file manipulation:

- `f_open(const char * fname, int mode)` (U) open a file name `fname` with the mode `mode` and return a file descriptor. The allowed modes are as follows: `F_WRITE` - writing and reading, truncates if the file exists, or creates it if it does not exist; `F_READ` - open the file for reading *only*, return an error if the file does not exist; `F_APPEND` - open the file for reading and writing but does not truncate the file if exists; additionally, the file pointer references the end of the file. `f_open` returns 0 on success and a negative value on error.

- `f_read(int fd, int n)` (U) read `n` bytes from the file referenced by `fd`. On return, `f_read` returns the number of bytes read, 0 if `EOF` is reached, or a negative number on error.

- `f_write(int fd, const char * str, int n)` (U) write `n` bytes of the string referenced by `str` to the file `fd` and increment the file pointer by `n`. On return, `f_write` returns the number of bytes written, or a negative value on error.

- `f_close(int fd)` (U) close the file `fd` and return 0 on success, or a negative value on failure.

- `f_unlink(const char * fname)` (U) remove the file

- `f_lseek(int fd, int offset, int whence)` (U) reposition the file pointer for `fd` to the `offset` relative to `whence`. You must also implement the constants `F_SEEK_SET`, `F_SEE_CUR`, and `F_SEEK_END`, which reference similar file `whence`s as their similarly named counterparts in `lseek(2)`.

You may require kernel level functions as well. Be sure to document them in your code and `README` file.

1.7 Shell

Once PennOS is booted (i.e., executed from the host OS), it will execute a shell. You will program this shell using the PennOS user interfaces described above. Although there is not strong separation between user and kernel land, **you may only use the user level functions to program your shell**. That is, you may only use functions indicated with a (U).

The shell is like any other `ucontext` thread running in PennOS and should be scheduled as such, except it will always have a priority level of -1. Unlike traditional shells, it is not capable of running arbitrary programs; instead you will provide built-in programs to execute within a user context. Below are the following features your shell should provide:

- *Synchronous Child Waiting*: PennOS does not provide a means to perform asynchronous signal handling; instead, you will use a synchronous signal handler. Before prompting for the next command, your shell will attempt to wait on all children using `p_wait`.
- *Redirection*: Your shell must handle >, <, and >> redirection; however, you are not required to have pipelines.
- *Parsing*: You must parse command line input, but you may use your previous implementation, or the parser provided in the previous project.
- *Terminal Signaling*: You should still be able to handle signals like `CTRL-Z` and `CTRL-C`, and neither should stop nor terminate PennOS. Instead, they must be properly relayed to the appropriate thread via the user land interface. PennOS should shutdown when the shell exits (e.g., when the shell reads `EOF`).
- *Terminal Control of* `stdin`: Just as before, you must provide protection of `stdin`; however, you cannot use `tcsetpgrp(2)` since PennOS executes as a single process on the host OS process. Instead, your OS should have a way to track the terminal-controlling process. Functions that read from `stdin` (e.g., `cat`), should be stopped (by them sending a `S_SIGSTOP`) signal if they do not have control of the terminal.

1.7.1 Shell Built-ins

The following shell built-ins should run as independently scheduled PennOS processes (indicated by (S*)). Additionally, you must provide three of your own built-in commands of your own choosing. The only requirement is that they must use a call to the user level interface and be reasonably interesting. Describe these three commands in your `README` and companion document.

- `cat [fname]` (S*) cat a file to `stdout`
- `nice priority command [arg]` (S*) set the priority level of the `command` to `priority` and execute the command
- `sleep n` (S*) sleep for `n` seconds
- `busy` (S*) busy wait indefinitely
- `ls` (S*) list all files in the file system and relevant information
- `touch file` (S*) create a zero size file `file` if it does not exist
- `rm file` (S*) remove a file `file`
- `ps` (S*) list all running processes on PennOS

The following shell built-ins should be run as the shell; that is, they should each execute as a shell sub-routine rather than as an independent process.

- `nice_pid priority pid` (S) adjust the nice level of process `pid` to priority `priority`
- `man` (S) list all available commands

- `bg [pid]` (S) continue the last stopped thread or the thread `pid`
- `fg [pid]` (S) bring the last stopped or backgrounded thread to the foreground or the thread specified by `pid`
- `jobs` (S*) list current running processes in the shell
- `logout` (S) exit the shell, and shutdown PennOS

1.8 PennOS Logging

You are required to provide logging facilities for your scheduler and kernel level functionality. The logs you produce will be used for grading, but you should also use the logs as part of your debugging and development. Any additional logging events you report should fit the formats below and should be documented in your README and companion document.

We will also provide a python script that parses your log files and reports some vital information. You may edit and modify this script as you need, but be sure to document such changes in your README.

All logs will have the following set format:

```
[ticks] OPERATION ARG1 ARG1 ...
```

Where `ticks` is the number of clock ticks since boot, `OPERATION` is the current scheduling procedure (e.g., swapping in a process), and the `ARGS` are the items being acted upon in the procedure. Your log file should be **tab delimited**. The following events should be logged with the following formats.

- Schedule event: The scheduling of a process for this clock tick

  ```
  [ticks] SCHEDULE PID QUEUE PROCESS_NAME
  ```

- Process life-cycle events: The creation and completion stages possible for a process.

  ```
  [ticks] CREATE PID NICE_VALUE PROCESS_NAME
  [ticks] SIGNALED PID NICE_VALUE PROCESS_NAME
  [ticks] EXITED PID NICE_VALUE PROCESS_NAME
  [ticks] ZOMBIE PID NICE_VALUE PROCESS_NAME
  [ticks] ORPHAN PID NICE_VALUE PROCESS_NAME
  [ticks] WAITED PID NICE_VALUE PROCESS_NAME
  ```

  Regarding process termination states, if a process terminates because of a signal, then the `SIGNALED` log line should appear. If a process terminates normally, then the `EXITED` log line should appear.

- Nice event: The adjusting of the nice value for a process

  ```
  [ticks] NICE PID OLD_NICE_VALUE NEW_NICE_VALUE PROCESS_NAME
  ```

- Process blocked/unblocked: The (un)blocking of a running process

  ```
  [ticks] BLOCKED PID NICE_VALUE PROCESS_NAME
  [ticks] UNBLOCKED PID NICE_VALUE PROCESS_NAME
  ```

- Process stopped/continue: The stopping of a process

```
[ticks] STOPPED PID NICE_VALUE PROCESS_NAME
[ticks] CONTINUED PID NICE_VALUE PROCESS_NAME
```

1.9 Arguments to PennOS

```
PennOS flatfs [schedlog]
```

If schedlog is not provided, logs will be written to a default file named `log` that will be created if it does not exits. Note that this log file lives on the host OS and not within a `flatFAT`. The `flatfs` argument is required and refers to the host OS file containing your FlatFAT file system.

2 Acceptable Library Functions

In this assignment you **are not** restricted in your use of library functions. Of course, if you use functions that are not in the spirit of this assignment, e.g., `fork(2)`, then you will receive a ZERO.

3 Error Handling

As before, you must check the error condition of all system calls for the host OS. Additionally, you must provide an error checking mechanism for the PennOS system calls. You may find it useful to leverage `perror(3)` mechanisms by setting the global `errno` value appropriately depending on the errors you encounter. **In your companion document, you must describe the `errno` values each system call could set.**

4 Companion Document

In addition to the standard `README` file, you are also required to provide a companion document for your PennOS. This document will contain all the documentation for all PennOS system calls, their error conditions, return values, etc. You can consider the man pages as a guide for the kind of information expected in your companion document. You should have the companion document completed by the milestone meeting, and **we will only accept submission of companion documents in PDF format**. If you submit in any other format, you will **receive a 0 on that part of the project**.

5 Memory Errors

Unfortunately, `valgrind` and `ucontext` do not play nice together; however, this does not mean you should not check your code for memory leaks and violations. As before, **code with memory leaks and memory violations will be subject to deductions**.

6 Developing Your Code

CIS 380 students are notorious for crashing *eniac* in creative ways, normally via a "fork-bomb." This always seems to happen about two hours before a homework deadline, and the result is chaos, not just for our class, but for everyone else using *eniac*.

This year CETS has asked us to lay out some guidelines. Really just one guideline: **Do not develop your code on *eniac***. Instead you should use the SpecLab for development. SpecLab is a collection of older desktops that run the same linux variant as *eniac*, and most importantly, you can crash them as much as you want without causing too much chaos.

You access SpecLab remotely over *ssh*. There are roughly 50-60 SpecLab machines up at any time. When you are ready to develop, choose a random number between 01 and 50, or so, and then issue this command:

```
bash# ssh specDD.seas.upenn.edu
```

where `DD` is replace with the number you chose. If that machine is not responding, then add one to that number and try again. Not all SpecLab machines are currently up, but most are.

You must be on SEASnet to directly *ssh* into SpecLab. The RESnet (your dorms) is not on SEASnet, but the machines in the Moore computer lab are. If you are not on SEASnet, you may still remotely access SpecLab by first *ssh*-ing into eniac and then *ssh*-ing into SpecLab.

You may develop your code on your personal machine. If you run a Unix variant, such as Ubuntu Linux, then you can develop your code locally. Although Mac OSX is BSD-like, there are enough differences in OSX that we advice against developing on it. More explicitly, the `ucontext` library does not work on Macs. Regardless of where you develop your code, it will be graded on the SpecLab Linux installation, and your program must work as specified there.

7 What to turn in

You must provide each of the following for your submission, *no matter what*. Failure to do so may reflect in your grade.

1. `README` file. In the `README` you will provide
   - Your name and `eniac` username
   - A list of submitted source files
   - Extra credit answers
   - Compilation Instructions
   - Overview of work accomplished
   - Description of code and code layout
   - General comments and anything that can help us grade your code
2. Companion Document describing your OS API and functionality. **Only PDF submissions will be accepted**.
3. `Makefile`. We should be able to compile your code by simply typing `make`. If you do not know what a `Makefile` is or how to write one, ask one of the TA's for help or consult one of many on-line tutorials.

4. Your code. You may think this is a joke, but at least once a year, someone forgets to include it.

8 Submission

Submission will be done electronically via *eniac*'s `turnin` command. `turnin` will not work from one of the lab machines in Moore 100 or on the SpecLab machines. **You must be logged into *eniac***. This should not be an issue because your home directory is NFS mounted.

To submit, place **all** relevant code and documents in a directory named, *username-username-username-username-project2*. **You must organize your code into directories as follows:**

- **project2/bin** - all compiled binaries should be placed here
- **project2/src** - all source code should be placed here
- **project2/doc** - all documentation should be place here
- **project2/log** - all PennOS logs should be placed here

Your code should compile from the top level directory by issuing the `make` command.

To submit your code, issue this command on the directory:

```
turnin -c cis380 -p project2 username-username-username-username-project1
```

9 Grading Guidelines

Each group will receive three grade entries for this project: A pass/fail grade for the project milestone and demo, and a numeric grade for the development. Poor performance in the milestone or demo may affect your numeric grade. In particular, we are expecting that your group has made significant progress by the milestone. That is, you should be more than 60% complete on each of the project parts and have PennOS code that runs and at least schedules dummy programs..

Each team will receive a group grade for the development; however, individual grades may differ. This can occur due to lack of individual effort, or other group dynamics. In 99% of cases, everyone on a team will receive the same grade. Below is the grading guideline.

- 10% Documentation
- 45% Kernel/Scheduler
- 30% File System
- 15% Shell

Please note that general deductions may occur for a variety of programming errors including memory violations, lack of error checking, poor code organization, etc. Also, do not take the documentation lightly, it provides us (the graders) a road-map of your code, and without it, it is quite difficult to grade the implementation.

Your programs will be graded on the SpecLab machines, and must execute as specified there. Although you may develop and test on your local machine, you should always test that your program functions properly there.

10 Attribution

This is a large and complex assignment, using arcane and compactly documented APIs. We do not expect you to be able to complete it without relying on some outside references. That said, we do expect you to struggle a lot, which is how you will learn about systems programming, by *doing things yourself*.

The primary rule to keep you safe from plagiarism/cheating in this project is to attribute in your documentation any outside sources you use. This includes both resources used to help you understand the concepts, and resources containing sample code you incorporated in your shell. The course text and APUE need only be cited in the later case. You should also use external code sparingly. Using most of an example from the `pipe(2)` man page is ok; using the `ParseInputAndCreateJobAndHandleSignals()` function from *Foo's Handy Write Your Own Shell Tutorial* is not (both verbatim, and as a closely followed template on structuring your shell).

A User Context "Hello World"

```c
#include <ucontext.h>
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define STACKSIZE 4096

void f(){
  printf("Hello World\n");
}

int main(int argc, char * argv[]){

  ucontext_t uc;
  void * stack;

  getcontext(&uc);

  stack = malloc(STACKSIZE);

  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = 0;

  sigemptyset(&(uc.uc_sigmask));

  uc.uc_link = NULL;

  makecontext(&uc, f, 0);

  setcontext(&uc);
  perror("setcontext"); //setcontext() doesn't return on success

  return 0;
}
```