POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

iSpy: A Real-Time Application of Shoulder Surfing Attacks

Thesis Supervisor: Federico Maggi

Thesis Co-Supervisor: Stefano Zanero

Naz Saltoglu

764823

Contents

Listings

Chapter 1

Introduction

As far as all the digital and physical ways of obtaining information about a person, spying on a person is the most reliable way of doing so. Even if the person is protected from all digital attacks by protection software, he/she has very little to do against being spied on.

The mobile application I developed focuses on the idea of digitalising and automating these kind of spying(or shoulder-surfing attacks). It is based on another eavesdropping attack[1], where they implemented the idea of eavesdropping on the input that is being entered to an iPhone, exploiting the magnification of the keys on the keyboard, with a video that is post processed, yielding reliable results. My application focuses on providing these results in real time on a mobile platform, with only the power of another mobile device. This way, it increases the usability and mobility of the attack, as it may be performed while the victim is unaware that such an attack is being performed on him/her.

The application relies on the computer vision and image processing techniques. With the help of the in-built camera of the device, it inputs a video to send to processing. Based on three different stages of image processing, it first detects the screen, distinguishes and separates it from the rest of the frame that is being captured by the camera. As the screen of the phone is probably tilted, or captured from an angle, the screen is transformed by the help of a feature detection algorithm and rectified to make use of the input. After this process, the screen goes through several processes to distinguish the background (i.e., the keyboard without any input or occlusion) from the fluctuations that may happen on the keyboard. These may be the enlarged keys or occlusions (fingers on the keyboard, etc). Later these fluctuations are processed to distinguish the actual input (enlarged keys) from occlusions. After these eliminations of the input, the application matches the template of the keys to the input according to their position on the keyboard. The user sees the estimated keys through the application in-real time, as well as the processed frames.

My implementation process was done with C++ and Objective-C, which uses the camera input to get the each frame, forward it to implementations of SURF algorithm to obtain the screen, plot the homography to rectify it, see if this yields a solution that corresponds with the optical flow of the screen, get the result and try to see if it yields a foreground which can be used by the image processing methods. If so, continue with thresholding the image to make use of the high pass filters and feed to the blob detectors that detect keys according to their size and area. Later on, finding the location of the key and matching the template of the key to the found area, to make sure we have the right output.

The tests were done with a random text and with changing conditions(lighting, angles, threshold of the blob finding and sampling frame rate). These conditions show to be in a very delicate equilibrium since with different settings, the correctly detected key rate may change very quickly. They show that the sampling rate is still not enough for recording all keystrokes, while from the ones that the device can distinguish, given the right conditions, the application is able to distinguish the keys at a decent success percentage at quasi real-time.

As this application is quite the first step for such shoulder surfing attacks via mobile, this is a promising one in mobile shoulder surfing attacks, while there is a lot of room to improve.

Chapter 2

Application Overview

2.1 Background and Theory

The architecture of the application is composed of the User Interface of the application, the View Controller, and the Image Processing Part, which could be seen in Figure 2.1. User Interface interacts with the View Controller, while View Controller interacts both with User Interface and Image Processing part.

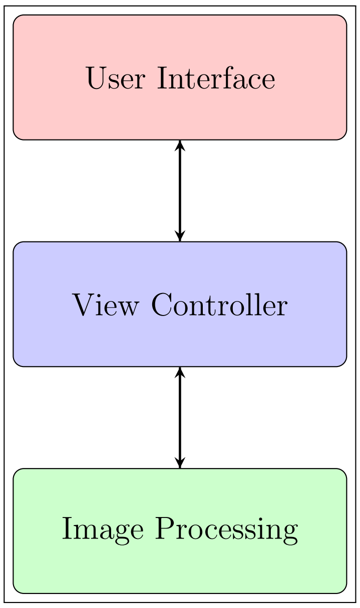


Figure 2.1: General Overview

The earlier work [1] focuses on creating a system that eavesdrops on iPhones exploiting the key magnification phase. They analyze the input video searching for a (possibly) tilted, distorted, or rotated image of the screen. The next step is to rectify and straighten the screen, making it easier to perform image processing on. Most of my work corresponds with their work throughout the image processing parts, while I had some parts to implement differently, given that OpenCV or Objective-C implements things differently, or was infeasible to implement, given that frames are reached in a different way. Their work was mainly based on Matlab, and was including the pre processing of a video with rectified frames, which were fed to the background later on. A real-time approach made background estimation and such other calculations based on previous frames a little more difficult, though I tried to implement each step of the algorithm that they have created in a another manner.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \backsim H' \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

The linear transformation that corresponds to the relation between the distorted screen and rectified screen is called Homography. Homography facilitates the re-alignment of the screen with an equation such as this: where (x', y') and (x, y) are the image coordinates of the points of the acquired images before and after rectification, respectively. 3 x 3 matrix H represents the homography relating the two images. The template image is used as a reference for after rectification screen, so that we can also exploit the common parts in both matrices and find the corresponding points to have a more solid estimation for homography matrix. When a screen is detected its image is tracked along the subsequent frames, following the natural movement of the user or of the spying camera. Then, a geometrical transformation is estimated to rectify the image of the screen thus eliminating distortions such as rotations or perspective deformations. The resulting image is almost equivalent to a screen template.

The rectification of the screen and homography calculation phase is almost the same as their work, though the rest is different as it is implemented in OpenCV and C++. These steps can be roughly seen in the Figure 2.2.

Phase 2 subtracts the background (i.e., an image of the virtual keyboard with no keys pressed) from each frame, to highlight the variations on the screen. For this background to be detected properly, we use the weighted background accumulation method. Every frame obtained adds to the background with a small weight. Let B_{t} be the background image at time t:

$$B_{\mathrm{t}} = (1 - \alpha)B_{\mathrm{t\text{-}1}} + \alpha F_{\mathrm{t}} \tag{2.1}$$

where F_{t} is the rectified frame at time t, and $B_{\mathrm{t\text{-}1}}$ is the background image at time t-1 and $\alpha \in$ [0,1] is an update parameter.

Equation 2.2 demonstrates how the accumulated background is subtracted from the frame to obtain the foreground at time t(Z_{t}).

$$Z_{\mathrm{t}} = |F_{\mathrm{t}} - B_{\mathrm{t}}| \tag{2.2}$$

The foreground consists of magnified keys, other objects such as fingers or other occlusions, or sometimes parts of the keyboard, given the change in lighting, angle or a rectification error may produce a difference in the location of the keyboard in

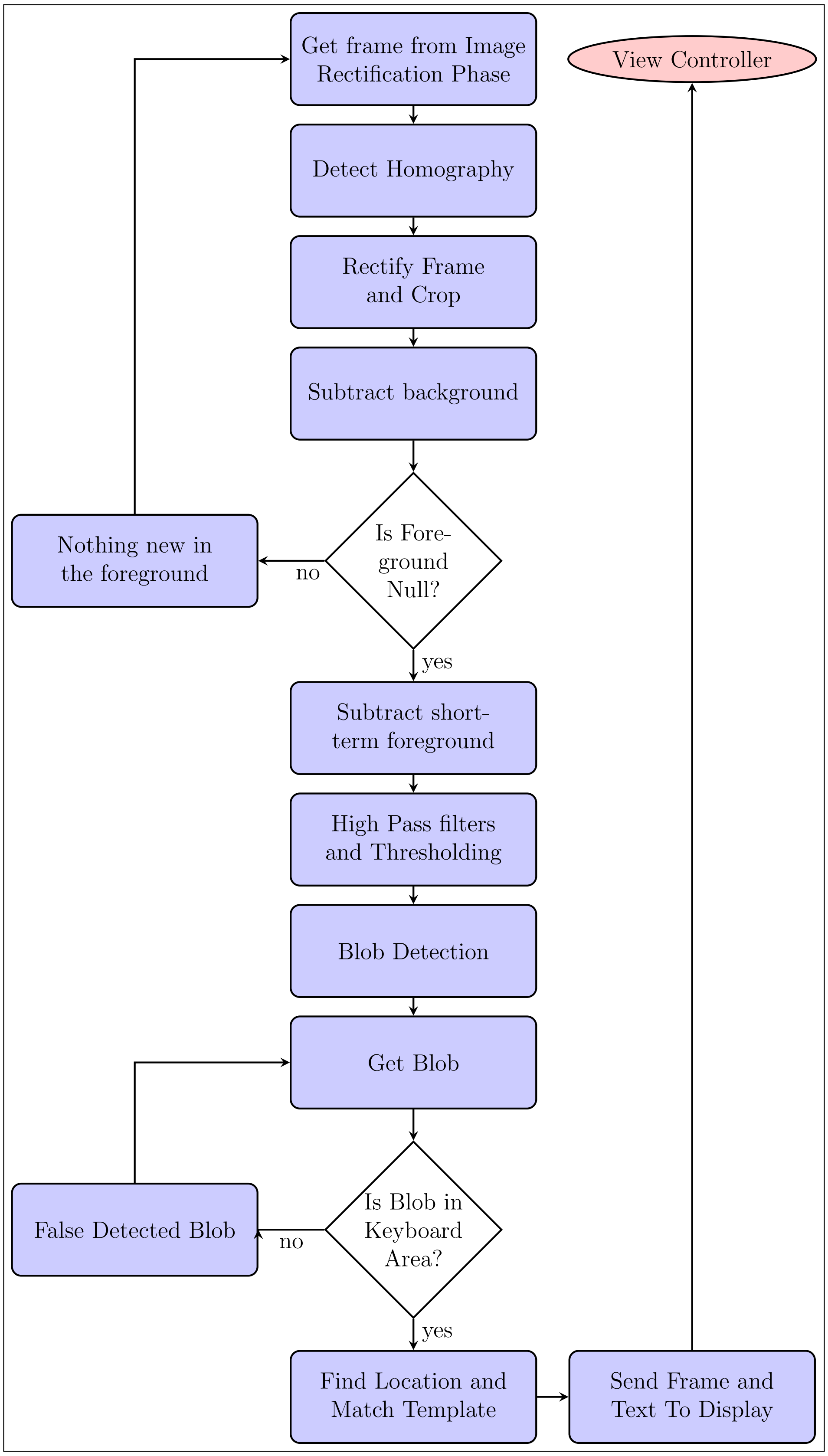

Get frame from Image Rectification Phase
View Controller
Detect Homography
Rectify Frame and Crop
Subtract background
Is Foreground Null?
no
Nothing new in the foreground
yes
Subtract short-term foreground
High Pass filters and Thresholding
Blob Detection
Get Blob
Is Blob in Keyboard Area?
no
False Detected Blob
yes
Find Location and Match Template
Send Frame and Text To Display

Figure 2.2: Phase 2 and 3 Overview

the frame and the background. Foreground is processed with high pass gradient and Laplacian filters to exploit the contrast of the keys and bring out the foreground contrasts in a better way. Equation 2.3 shows how the output image(dst) is being created:

$$\texttt{dst} = \Delta\texttt{src} = \frac{\partial^2\texttt{src}}{\partial x^2} + \frac{\partial^2\texttt{src}}{\partial y^2} \tag{2.3}$$

After these processes, we blend the output of the Laplacian method with the short term foreground, and we apply a threshold value to distinguish between the blobs that yield magnified keys,other possible blobs and rest of the image for good.

$$dst(x, y) = \begin{cases} 0 & \text{if src(x,y)>threshold} \\ \text{maxVal} & \text{otherwise} \end{cases}$$

These variations are either fingers, removed with appropriate image filtering techniques, or the visual feedback we want to capture. In Phase 3, the recognition phase, the center of each blob (highlighted area) is computed, and matched to the keyboard layout to determine the general area of the pressed key. Then, the templates of the letters neighboring the target region are exploited to find the best-matching areas, thus recognizing the keystrokes (if any).The best-matching key is looked up within those candidate letters that have a percentage of black and white pixels similar to their templates.

The phases 1-3 determine the best-matching key at each frame i.e., the magnified key that has been most likely pressed in each frame.

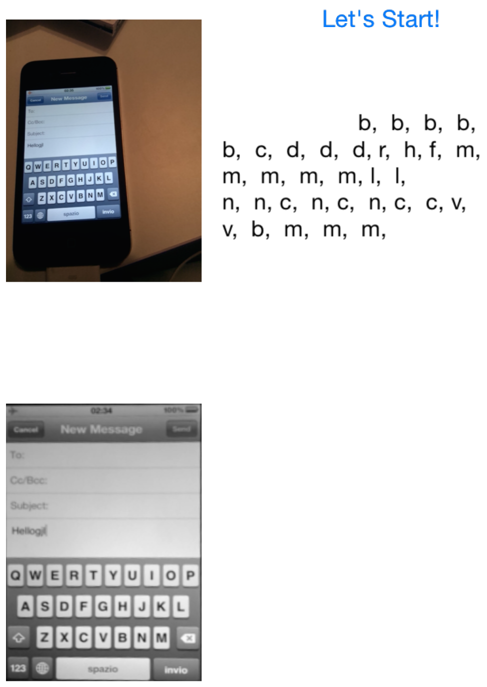
2.2 User Interface

The Main View of the application consists of many different kind of objects:

- Two Image Views(UIImageView): One for displaying the Camera Input, the other one for displaying the output of the processed frame.
- One text display(UITextView) for displaying the estimated keys that are pressed
- A "Start" button that starts the input flow of frames to the View Controller.

2.3 View Controller

The View Controller is the vital key control mechanism between the application's data and its visual interface. The initialization and configuration of the user interface objects is managed through View Controller. Every frame or text goes through View Controller to get displayed. Also the events that are triggered by the user via gesture, tap or button clicks get managed in View Controller to execute the appropriate action and managing the functional part of the code.



b, b, b, b,
b, c, d, d, d, r, h, f, m,
m, m, m, m, l, l,
n, n, c, n, c, n, c, c, v,
v, b, m, m, m,

Figure 2.3: View of the User Interface

In our application, we use the View Controller for:

Setting up the Camera(device) and Video Session Configurations: This process takes place when the app is invoked. We ensure that the video input session is initialized and setup properly. Later we move on to setting up it's quality of the frames that will be flowing through. Moreover, we ensure that the Capture Session is created and configured properly.

Pressing the Start Button and Initialising the Output : Pressing the "Start" button on the User Interface triggers the output setup and input flow. While setting up the output flow, we make sure that the system does not dispose of the frames that wait while the other frames are being processed. This is a crucial part of our operation since our processing procedures take relatively longer than the speed of the frames being buffered. Therefore, our frames wait a bit to get processed, as it is essential for our process that they do not get discarded.

Capturing Output and Passing Frames to Image Processing: As session starts running, output captures frames of the camera input and loads them into a buffer. Each time the output captures frames, a method is invoked to notify the user. We use this method to obtain the frame from the buffer and then pass this frame to the Image Processing part to start the processing immediately. The Image

Processing part sends back the processed frame and the estimated keys, and we use our dispatch queue to display them in the correct order. If the output drops samples (frames) for some reason, the user is notified by another method. We use it to investigate why the frame has been dropped.

2.4 Image Processing

2.4.1 Phase 1: Screen Detection and Rectification

This phase is divided into two sub-tasks executed in cascade: screen detection, that searches for any occurrence of the screen in the input video, and image rectification and crop, which estimates the perspective difference of the detected screen,rectifies and crops its image accordingly. Both methods rely on feature extraction and matching: an image feature is a small image patch centered on a point of the image, usually where the image presents a discontinuity, e.g., a corner or an edge. Given two images and their features, the features can be matched in order to find image correspondences. In our work we use features which are invariant to rotation, scaling and skew transformations.

Screen Detection: To detect the screen in the frame is a quite costly and challenging task since the perspective, the size and the position of the screen in the frame can vary as the camera moves. Therefore, the whole frame must be searched for the screen image. Also, the screen can be occluded by fingers or other unknown objects.

For these reasons, we use a feature-based template matching algorithm, SURF. The features of the template are matched with the features of the frame,in order to find the corresponding points and detect the screen's location within the frame. These matchings may sometimes provide false matching points, those mismatches are eliminated within the Homography estimation process, which we will discuss on the Screen Rectification phase. The output of this phase is the corresponding points of the screen inside the frame.

Screen Rectification: Input of this phase is indeed the output of the Screen Detection phase. The corresponding points, which are aligned in an angle, are distorted, and they need to be realigned to sustain the original perspective of the screen. Homography creation also provides the functionality to discriminate inliers and outliers i.e. good and false points. If inliers outnumber the outliers, the estimated homographs is used for rectification of the image. Otherwise, the homography is considered to be false, as if no matches were found, and therefore not used, while the frame is discarded. This gives the chance to eliminate false positives provided by the Screen Detection phase.

Finally the estimated homography is used to rectify and crop the screen accordingly. The output of this phase is the rectified and cropped screen, providing a more polished and ready view for background estimation and subtraction. Figure 2.4 shows the outcome of this phase.

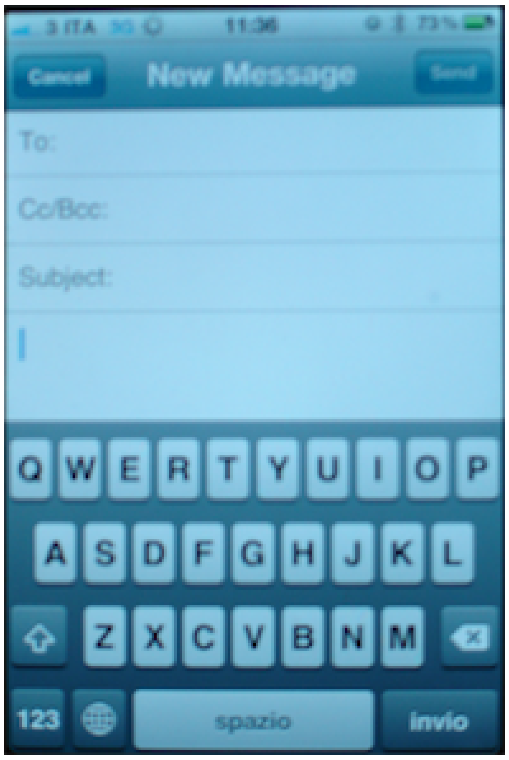


Figure 2.4: Output of Screen Detection and Rectification

2.4.2 Phase 2: Background Subtraction and Edge Detection

After receiving the rectified and cropped frame, we perform a background subtraction to highlight the dynamic changes on the screen (foreground), whether it be fingers or magnified keys.

For this background to be detected properly, we use the weighted background accumulation method. Every frame that we obtain adds to the background with a small weight. The accumulated background is differenced from the frame to obtain the foreground. The foreground is checked to be empty or not, since if it is empty, it means that we cannot identify any magnified keys. If it is not, we continue with the process.

The foreground consists of magnified keys, other objects such as fingers or other occlusions, or sometimes parts of the keyboard, given the change in lighting, angle or a rectification error may produce a difference in the location of the keyboard in the frame and the background. We pass the foreground through high pass filters to exploit the contrast area of the keys. Foreground is processed with Sobel and Laplacian filters, the conclusion becoming a mixture of both.

After these processes, we blend the output of the Laplacian method with the short term foreground, and we apply a threshold value to distinguish between the magnified keys and rest of the image for good. If there is a pixel that is over a decided threshold value, it will be assigned the color white, while all the pixels that have a value below the threshold will be assigned black. This will help us in binarizing the image.

In addition to this, the binary image goes through a morphological transformation that is called "closing", for eliminating the noise - or unnecessarily detected pixels. The objective is to get rid of all the noise of the smaller pixels that may have passed through the high-pass filters and thresholding. Finally, the last step of our foreground extraction and elimination: the edge detection. For edge detection we use the Canny algorithm, then we find and trace the extracted edges on to the frame by only tracing the external parts of the detected regions.

2.4.3 Phase 3: Magnified Keys Detection and Template Matching

In this phase, we detect the blobs that may be the possible magnified keys and filter them by their area and their color, and get the centres of the blobs that may be candidates to be magnified keys. Our next step is to try to confine them according to their place on the screen, since we only need the blobs that occur in the keyboard area. First elimination comes through the acceptance of the blobs that are only in the keyboard area, later the blobs are distinguished according to their rows, given that we have three rows on the keyboard to choose from (the last row is not considered, see Section 4). After that according to their locations on the row, their location detection is made. The location of the keys lead to the estimation of which key was pressed and matching the template of that key to the estimated key.

Chapter 3

Implementation Details

3.1 User Interface

The Interface of the application is created with the help of the Storyboard, which lets the users to use a graphical user interface while creating the interface of the application, instead of coding it in. The storyboard design of the application may be seen in Figure 3.1.

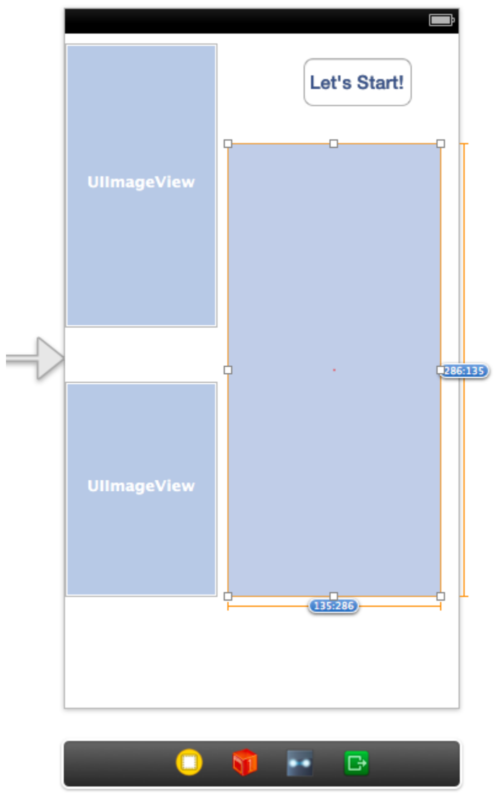


Figure 3.1: View of the Storyboard

3.2 View Controller

Setting up the Camera(device) and Video Session Configurations: When we open the app, we make sure that all the configurations for the Camera and Input session is set up correctly.

We set the quality of the frames Medium: High quality seems to have a negative effect on the memory that the app uses, while low has too little resolution for the image processing to run smoothly. Since it is not set to a constant number, but a generic value "Medium", it will be different on each device, depending on their performance.We make sure that the camera is setup, and will only input in video type. The next step is to make sure that the input coming from the camera is setup, and displayed in the UI. If we have any kind of device error, NSLog keeps track of the issues. The Initialization Snippet demonstrates how all these parts come together:

Listing 3.1: Initialization Snippet

```

    [super viewDidLoad];
    session = [[AVCaptureSession alloc] init];
    session.sessionPreset = AVCaptureSessionPresetMedium;

    CALayer *viewLayer = self.frameView.layer;

    AVCaptureVideoPreviewLayer *captureVideoPreviewLayer =
        [[AVCaptureVideoPreviewLayer alloc] initWithSession:session];

    captureVideoPreviewLayer.frame = self.frameView.bounds;
    [self.frameView.layer addSublayer:captureVideoPreviewLayer];

     AVCaptureDevice *device = [AVCaptureDevice
        defaultDeviceWithMediaType:AVMediaTypeVideo];
    NSError *error = nil;
    input = [AVCaptureDeviceInput deviceInputWithDevice:device error:&error];
    if (!input) {
        NSLog(@"ERROR: trying to open camera: %@", error);
    }
```

For the Capture Session to start, we add the input coming from the camera to the session and start the configurations. We set the frame rate of the session as we please. Our tests include experimenting with different frame rates, for more information on them Chapter 5. Later on, if the Auto Focus mode is available on the testing device, we disable the option since it may cause null frames, by constantly moving in and out of focus., whereas disabling the option lets the camera use the autofocus feature only once, in the beginning of the session. Also White Balance Mode goes through the same procedure since if the option is kept on, constant white balancing of the screen when there is a change of lighting, may also cause null frames to be captured through output. After locking in the configurations, the session is started. Session Configuration Snippet demonstrates this configuration.

Listing 3.2: Session Configuration Snippet

```

[session addInput:input];
[device lockForConfiguration:nil];
[session beginConfiguration];
[device setActiveVideoMinFrameDuration:CMTimeMake(10, 300)];
[device setActiveVideoMaxFrameDuration:CMTimeMake(10, 300)];
NSLog(@"formats %@ ", device.activeFormat.videoSupportedFrameRateRanges);
if([device isFocusModeSupported:AVCaptureFocusModeLocked]){
       [device setFocusMode:AVCaptureFocusModeLocked];
       NSLog(@"Auto Focus Locked");
 }
if ([device isWhiteBalanceModeSupported:AVCaptureWhiteBalanceModeLocked]) {
       [device setWhiteBalanceMode:2];
       NSLog(@"White Balance Mode Locked");
}
[session commitConfiguration];
[device unlockForConfiguration];
[session startRunning];
```

Pressing the Start Button and Initialising the Output : Pressing the "Start" button on the User Interface triggers the output setup and input flow. First of all we create an output channel, and make sure that the video settings are arranged correctly. `setAlwaysDiscardsLateVideoFrames: NO` enables us to ask the system not to discard the late processed frames. An essential part of our process is our dispatch queue. A dispatch queue can be described as a task queue that executes the passed tasks onto it in the specified order. `setSampleBufferDelegate` helps us to illustrate which queue to set callbacks to, when a new video sample buffer is captured. It is sent to the sample buffer delegate using `captureOutput:didOutputSampleBuffer:fromConnection:`, which can be viewed in Capture Output Snippet. All delegate methods are invoked on the specified dispatch queue. We implement the initialization the connection as described in the Button Action Snippet:

Listing 3.3: Button Action Snippet

```

- (IBAction)processButton:(id)sender {
   dataOutput = [AVCaptureVideoDataOutput new];
   dataOutput.videoSettings = [NSDictionary dictionaryWithObject:[NSNumber
       numberWithUnsignedInt:kCVPixelFormatType_32BGRA] forKey:(NSString
       *)kCVPixelBufferPixelFormatTypeKey];
    [dataOutput setAlwaysDiscardsLateVideoFrames:NO];
   AVCaptureConnection *videoConnection = [dataOutput
       connectionWithMediaType:AVMediaTypeAudio];

   if ( [session canAddOutput:dataOutput])
       [session addOutput:dataOutput];

   dispatch_queue_t queue = dispatch_queue_create("VideoQueue",
       DISPATCH_QUEUE_SERIAL);
    [dataOutput setSampleBufferDelegate:self queue:queue];

   if (!running){
       running = YES;
       [session startRunning];
   }
   else{
       [session stopRunning];
       running = NO;
   }}
```

Capturing Output and Passing Frames to Image Processing: We use it to make sure the output frame and the estimated keys are being displayed in the synchronously. The `captureOutput:didOutputSampleBuffer:fromConnection:` method notifies the caller that a video frame is captured by output and written to sampleBuffer. This method includes the reference to the buffer containing the frame:

Listing 3.4: Capture Output Snippet

```

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
      fromConnection:(AVCaptureConnection *)connection
{
    UIImage *image = [self imageFromSampleBuffer:sampleBuffer];
    if (image!=NULL) {
        frame = [ImageProcessing imageProcess:image];
        NSString* candidates = [ImageProcessing getResult];
        NSLog(@"Frame processed.");
        dispatch_sync(dispatch_get_main_queue(), ^{
            _outLabel.text = candidates;
            [rectifiedView initWithImage:frame];
        });
    }
}
```

`imageFromSampleBuffer` method creates a UIImage from the Captured Output frame that was put in the buffer. `ImageProcess:` method obtains the output frame that includes the guessed blobs from the ImageProcessingPart. `getResult` method returns the text of estimated keys through all frames. We call the dispatch queue to display the text of keys, and the output frame to the user.

If the output drops samples(frames) for some reason, the user is notified by `captureOutput:didDropSampleBuffer:fromConnection:` method. We use it to investigate and log why the frame has been dropped.

3.3 Image Processing

While getting the frame from the View Controller and sending it back, I used another method to have all the settings ready for the code and type conversions of Objective-C to C++ and vice versa. `imageProcess` method helps the back and forth communication with View Controller and initializing the needed parameters. We get the screen template from our application and initialise it, to be used in rectification stage. We also get the image that View Controller passed us, do the conversion from UIImage to cv::Mat, and check if it has come NULL, if so, since we cannot do any processing on it, we send it back, if it is not NULL, we move on to the Image Processing phases. When the processed frame comes back, we send it back to the View Controller. imageProcess Method Snippet, goes in detail, on how the implementation for this is performed:

Listing 3.5: imageProcess Method

```

+ (UIImage*) imageProcess: (UIImage*) image
{

    NSBundle* bundle = [NSBundle mainBundle];
    NSString *imagePath = [bundle pathForResource:@"template" ofType:@"png"];
    NSData* data = [[NSData alloc] initWithContentsOfFile:imagePath];
    templateFile = [[UIImage alloc] initWithData:data];

    Mat mtx = [self cvMatFromUIImage:image];

    cvtColor(mtx, cur.img_proc, CV_BGR2GRAY);
    cur.img_out = 0;
    cur.img = mtx;
    if (cur.img.empty()){
        return NULL;
    }
    else
    {
        cv::Mat rectifiedFrame = processStepOne();
        UIImage* rectified = [self imageWithCVMat:rectifiedFrame];
        return rectified;
    }
}
```

3.3.1 Phase 1: Screen Detection and Rectification

During screen detection and rectification phase, the crucial parts of implementation is the way homography is calculated and the screen is rectified.

The first step to calculate homography is to use the SURF algorithm to extract keypoints and descriptors for both template of the screen and frame that includes the screen.

Keypoints include the location of the points that get over the Hessian threshold. The Hessian threshold determines how large the output from the Hessian filter must be for a point to be used as an interest. Theoretically, bigger the Hessian threshold gets, the less points (although more reliable) we get. We implement keypoint extraction as follows:

Listing 3.6: Extracting Keypoints Snippet

```

void frame_obj::Extract_surf_keypoints(){
   int minHessian = 600;
   std::vector<KeyPoint> keypoints;
   SurfFeatureDetector detector( minHessian );
   detector.detect( img_proc, keypoints );
   key=keypoints;
}
```

Descriptors, on the other hand, enables us to compare the keypoints that we have obtained. It is size, perspective and shade independent. Therefore if we obtain the descriptors for both screen template and the frame that includes the screen, we would be able to comment on how similar or different they are by using the descriptors.

Listing 3.7: Extracting Descriptors Snippet

```

void frame_obj::Extract_descriptor(){
   SurfDescriptorExtractor extractor;
   Mat descriptors;
   extractor.compute( img_proc, key, descriptors );
   desc=descriptors;
}
```

If the extraction processes go without any problems for both template and frame, we estimate the homography via the findHomography function. During this process we also distinguish between good matching points and false matching points.

Listing 3.8: Homography Estimation Snippet

```

for( int i = 0; i < matching.size(); i++ )
       {
       //-- Get the keypoints from the good matches
       obj.push_back( keypoints_obj[ matching[i].queryIdx ].pt );
       scene.push_back( key[ matching[i].trainIdx ].pt );
       }
       Mat H = findHomography( obj, scene, CV_RANSAC, 3 ,mask);

       int cont_inliers=0;
       int cont_outliers=0;
       for(unsigned int k=0; k<obj.size();++k){
              if(mask.at(k)) {
                      ++cont_inliers;
                }
               else {
                      ++cont_outliers;
                 }
        }
       inliers=cont_inliers;
       outliers=cont_outliers;
       return(H);
       }
```

After a homography is estimated, we check to see the optical flow of the homographies, how this new homography compares to the one that was detected before. When a screen is detected its image is tracked along the subsequent frames, following the natural movement of the user or of the spying camera. This comes from the theory that even if there is a camera movement, it is generally not sudden but follows a natural pattern. For example, if the camera is moving right a bit, the next frame will move right a bit too, or at least will be close to the last frame location that was detected. With this method we ensure that our frame rectifications are running smoothly.

We create the homography according to the optical flow. If everything is OK, we go on to rectifying the screen via warpPerspective method. The correct perspective is provided by the newly computed Homography matrix. After it is rectified, the frame is cropped according to the location and size of the phone screen. The Rectification Snippet shows how they are implemented:

Listing 3.9: Rectification Snippet

```
warpPerspective(img_frame,crop,Homo.inv(DECOMP_LU),
    cvGetSize(img_template_color));

```

```
Mat cropedImage (img_template_color,true);
crop.copyTo(cropedImage);
scene_corners.clear();
```

The output of this phase is the cropped and rectified screen to be fed to the background subtraction.

3.3.2 Phase 2: Background Subtraction and Edge Detection

For Background Subtraction phase, we obtain the proper frame from the rectification step, and we take the absolute difference of the old background from the current frame, to visualise the foreground. If there is nothing on the foreground, we have no new data to process, so we return the frame.If there is more data on the foreground, we use the weighted background accumulation method via matrix multiplication with a scalar. Note that the following piece of code implies the rest of the image processing material all work under the assumption that foreground has some data to work with and the frame is different from the background:

Listing 3.10: Background Subtraction Snippet

```

absdiff(frame, background, fore);
if (fore.empty()) return frame;
background = (alpha * frame) + ((1-alpha) * background);
```

with the weight alpha being 0.05. We would like very little influence of each frame to the background, assuming that the first frame we obtain is the proper background.

Later on we subtract the short term foreground from the actual foreground. We perform the same accumulation method as background for the short term foreground, with alpha being higher, given that we want the short term foreground to only last a few frames, while a key is magnified. Then we feed the output of that operation into the `Laplacian` method of OpenCV. That output is fed to the threshold, dilate and erode operations in the following way:

Listing 3.11: Thresholding and Morphological Filter Snippet

```
absdiff(fore, foreground, fore);
foreground =(0.5 * fore) + (0.5 * foreground);
Laplacian( fore, dst, ddepth, 3, scale, delta, BORDER_DEFAULT );
output = dst;
```

```
threshold(output, output, 30, 255, THRESH_BINARY_INV);
dilate(output,output,Mat());
erode(output,output,Mat());
```

3.3.3 Phase 3: Magnified Keys Detection and Template Matching

We first start with the detection of the blobs and retrieving their centers. SimpleBlobDetector class helps us in defining these blobs. The SimpleBlobDetector class is defined with parameters to our blobs are filtered by area and by color black, given that the outcome of the output should be black. The Blob Detection snippet shows how it has been initialised and detected.

Listing 3.12: Blob Detection

```

SimpleBlobDetector::Params params;
params.minDistBetweenBlobs = 20.0;
params.minThreshold = 50;
params.maxThreshold = 150;
params.thresholdStep = 5;
params.filterByInertia = false;
params.filterByConvexity = false;
params.filterByColor = true;
params.blobColor = 255;
params.filterByCircularity = false;
params.filterByArea = true;
params.minArea = 200.0;
params.maxArea = 1600.0;

SimpleBlobDetector * blob_detector;
blob_detector = new SimpleBlobDetector(params);
blob_detector->create("SimpleBlobDetector");
vector<cv::KeyPoint> keypoints;
blob_detector->detect(output, keypoints);
```

The matrix output is the matrix of the thresholded foreground to be detected, while detect method passes keypoints vector as an output that holds the centers of the blobs that have been found.

The detection of the position of the blob with the corresponding key is enforced by elimination of the blobs firstly through confining them the general keyboard area, then to rows, then to the exact position. The letters are distinguished by their rows according to the y-axis position of the blob center. Later on, each row Letters A, L , Q and P, are specifically found given that they are on the ends of the keyboard, the rest of the key centres are compared to the neighbouring key centers and found through absolute difference of both distances. In the Position Detection snippet, the detailed code is shown. `X` and `Y` refer to the x and y-axis coordinates of the center of the blob that has been detected, while `keyCenterCoordinates` is the vector of the centroids of each key on the template, stored accordingly to the QWERTY setting.

Listing 3.13: Position Detection

```

if (Y < (frameRows * 0.54)){
        if(X<(frameColumns*0.05))
                 letterNum=1;
        else if(X>(frameColumns*0.85))
                letterNum=10;
        else if((abs(keyCenterCoordinates[(X/17)].x - X)
                <(abs(keyCenterCoordinates[(X/17)+1].x - X))))
                 letterNum=(X/17);
        else letterNum = (X/17)+1;
 }
else if((Y < (frameRows * 0.66))&&(X <(frameColumns * 0.95))){
        if(X<25){
        letterNum=11;
        }
        else if(X>(frameColumns*0.85))
                 letterNum=19;
        else if((abs(keyCenterCoordinates[(X/19)+10].x - X)
                <(abs(keyCenterCoordinates[(X/19)+11].x - X))))
                letterNum=(X/19)+10;
        else letterNum = (X/19)+11;
 }
else if (X > (frameColumns*12/100) || (X < (frameColumns* 82/100)))
{
        letterNum = ((X)/(frameColumns*10/100))+19;
        if (letterNum>26) continue;
}
else continue;
```

After the detection of the position, it is time to see whether the blob we have found is a Magnified Key or not. We select a Region Of Interest (ROI) within the frame, which is approximately in the selected key's coordinates and shape. ROI is selected as follows: for the letters that are on the either side of the keyboard, a ROI is selected in a manner that preserves the frame boundaries, while for the other keys, it is a 40x40 square that takes the detected key center in the middle.

In the ROI Detection Snippet, the implementation details can be seen.

Listing 3.14: ROI Detection

```

if (keyCenterCoordinates[letterNum-1].x<25){
        roiRect = cv::Rect(0,keyCenterCoordinates[letterNum-1].y-20,40,40);
}
else if(keyCenterCoordinates[letterNum-1].x>155){
        roiRect = cv::Rect(keyCenterCoordinates[letterNum-1].x-20,
            keyCenterCoordinates[letterNum-1].y-20,
            frameColumns-keyCenterCoordinates[letterNum-1].x+20, 40);
}
else  roiRect = cv::Rect(keyCenterCoordinates[letterNum-1].x-20,
    keyCenterCoordinates[letterNum-1].y-20, 40, 40);
```

For this reason we compare the template of the key that has been found, versus the region of interest of the key. Then display the result if that key is correctly detected. This is done by the openCV's matchTemplate and the `minmaxLoc` methods, we compare our ROI with the detected location. minMaxLoc determines which is the spot that the matching has been done. Result Detection Snippet shows how this procedure is laid out:

Listing 3.15: Result Detection

```

matchTemplate( roi, templ, result, CV_TM_SQDIFF_NORMED );

normalize( result, result, 0, 1, NORM_MINMAX, -1, Mat() );

cv::Point minLoc; cv::Point maxLoc;
cv::Point matchLoc;

minMaxLoc( result, & minVal, & maxVal, & minLoc, & maxLoc, Mat() );

minLoc.x=minLoc.x + roiRect.x;
minLoc.y = minLoc.y + roiRect.y;
matchLoc = minLoc;

aa << letterNumArray[letterNum-1] << ", ";
rectangle( frame, matchLoc, cv::Point( matchLoc.x + templ.cols , matchLoc.y +
    templ.rows), Scalar::all(0), 2, 8, 0 );
```

Chapter 4

Testing

4.1 Requirements and Assumptions

The system is limited to the visual feedback of the magnified keys. There are a few situations where the keys may not be estimated:

- Lighting changes and angles,
- Sudden movements of the camera or the phone, or in general situations that may cause Homography estimation process to be wrong,
- Fingers occluding half or all of magnified key,
- The frame which includes the magnified key not being sampled.

I also consider the QWERTY keyboard only and and given that space key is not magnified, I did not include the detection of the space key within the algorithm. This also excludes the difference between lowercase and uppercase letters. Therefore, I assume that I can only consider the keyboards with a qwerty setting, and no space key or the punctuation, with case-insensitivity.

4.2 Tests Performed

The first test I made is about the performance of the application and how fast it displays the results back to the user. I made a test case where I pressed random letters, in an artificially lit environment with iPad Mini second generation and kept

Lighting	Threshold Range	Frame Rate	Magnified Input	Magnified Output	Difference	% Correct
Natural	50-150	30	”asdfghjkl”	”asdfghjk”	02.84’	88
Natural	100-200	30	”asdfghjkl”	”adfghkl”	02.66’	78
Natural	50-150	20	”ghjkl”	”ghjk”	04.35’	44
Artificial	50-150	30	”asdfghjkl”	”sfhjkl”	02.8’	66
Artificial	100-200	30	”asdfghjkl”	”adfghj”	02.30’	66
Artificial	50-150	20	”asdfghjkl”	”adfgjk”	02.53’	55

Figure 4.1: Test Results

track of the time of recognition between the time that I press the first letter on the keyboard and the time that we see the same frame on the output display.

Later on, I made another test where I changed the control elements a bit, lighting, moving the phone or the testing device and the threshold. The Random Text of ”asdfghjkl” was written every time.

4.3 Results

In my first test, the results showed that show that between the time that we press the first frame and the frame is displayed back, we have around 03’55 seconds in average. There were frames that were not displayed, or that my finger covered the magnified key. Those tests are not covered in the results and discussed further in the next section. The tests show that through continuously changing conditions, the performance changes as well. In my second controlled test, the Test Results table indicate how the test results came to be. Difference field in the table represents the time between the first frame is pressed on the keyboard and the frame is displayed back to the user. Magnified Input field is the keys that were readable from the sampled frames that have reached the Phase 2. Magnified Output field are the keys that were displayed back to the user. Therefore we aim to show how fast the frame is processed and returned to the user via User Interface. These tests are all done on one device, which is the iPad mini second generation.

I also did some tests on the Second Generation iPad, they proved to be very fruitless, given that the device’s computing power was too slow to handle the heavy-load image processing computations in real-time, which caused a lot of dropped frames and very little sampling rates, which gave us even less to work with in terms of blobs and magnified key samples.

4.4 Corner Cases

4.4.1 Lighting and Angles

The application is quite sensitive to even small light changes, which causes a change in the foreground. If the change in the foreground is effective enough to be able to pass through the Laplacian filters, thresholding and closing, they cause the application to recognise some blobs that are not there, which occasionally causes false estimations of the pressed keys.

The sudden change of the angles provide a disturbance in the homography estimation, which disturbs the image returned to the image processing part, which may cause false estimation of the key centers, therefore false matching of templates.

4.4.2 Frame Skipping and Sampling Issues

The tests also show that lower frame rates cause less sampling, therefore less magnified key containing frames. Though 30 frame per second is also not completely safe for not skipping samples. No matter how fast we set the frame rate to keep the sampling rate high, with the existing technology(using one of the fastest computing mobile devices of iOS), it seems like we are not able to prevent skipping of some frames and therefore magnified keys, given that sampling rate is still quite slow. This leads to less on-point recognization of the keys.

The inability to detect keys also occur when an object is intervening with the view of the magnified key, either partially or completely. These objects could be fingers which is often the case.

Another reason for this, is a badly detected rectified frame, even when there is no significant change of an angle or lighting, which causes the detection of no frames at all, or in some cases, false frames.

Chapter 5

Conclusions

I developed this application to demonstrate the feasibility of a real-time, automated shoulder surfing. I believe that I fulfilled that objective. This application makes the real-life scenarios of automated shoulder surfing quite real, given that all the attacker would need is a mobile phone with this application, and the victim's compatible phone. This application proves that spying on another person is quite easy and it will become easier as technology advances.

I described the mathematical background, general architecture and the implementation parts of the application that I have built, in-detail. I have shown that it can work efficiently and can recognise keystrokes in almost real-time, while maintaining a decent correctness percentage. I have met a lot of difficulties, such as lighting, sampling rates, angle of the devices and sudden movements, since there are a lot of performance-affecting conditions that need to be more stable. Obviously, the limitations to the device performances and the algorithm leaves space for improvement.

Admittedly, there are a lot of steps to go to estimate the pressed keys better, having stable performances even through angle changes and frame sampling, processing rates getting higher to enable higher correctly guessed key rates. Another branch of this application could be the estimation of the keys through the color changes of the pressed keys rather than magnifications, or a different estimation scheme with different keyboard layouts and key templates.

Right now, our assumptions that helped me build the application make it a tiny step in this big world of shoulder surfing.

Chapter 6

Bibliography

[1] Maggi, F., Volpatto, A., Gasparini, S., Boracchi, G.,Zanero, S. "A Fast Eavesdropping Attack Against Touchscreens" in Information Assurance and Security (IAS), 2011 7th International Conference on, 2011. p. 320-325. ISBN: 978-1-4577-2154-0 DOI: 10.1109/ISIAS.2011.6122840, 05 Dec 2011.