Biometric Systems - Term Paper
Smart Watch Prototype (SWP)

Stephan Thordal Larsen[1]

Abstract: ABSTRACT YADA YADA

Keywords: YADA, YADA, YADA

1 Introduction

The miniaturization of computing hardware which spawned the smartphone era, has also allowed for the development of powerful smartwatches filled with sensors. These sensors can continuously observe the wearers bodily functions, which in theory can be utilized for biometric identification of the wearer. This could allow for a strong link between the user and smartwatch, resulting in unobtrusive and ubiquitous user authentication.

This paper will analyse the first generation of smartwatches, more specifically the *Apple Watch*, to find which biometric capabilities that can be utilized for user identification and authentication. The analysis will look into the available sensors and the *Software Development Kit (SDK)* provided by Apple, in order to find which biometric measurements can be performed. The paper will try to identify both possibilities and limitations of the system, and try to utilize the possibilities in a prototype, to see how this in practice could be implemented.

The prototype will include feature extraction from biometric measurements and a comparison of these, in order to identify the wearing individuals. Lastly the prototype will be tested on 10 subjects, to evaluate the prototypes performance.

[1] DTU Compute, Denmark, s146907@student.dtu.dk

2 Analysis

This section will analyze the potential for capturing biometrics on the *Apple Watch*. The analysis will cover both the hardware and software, i.e. which sensors are included in the watch and how they are utilized by the provided frameworks. The sensors will also be evaluated on their usefulness for biometric identification. Finally the analysis should result in a selection of sensors found appropriate for the prototype.

2.1 Sensors

The *Apple Watch* includes a multitude of sensors [2], which are used mainly for usability, activity, fitness and health tracking, but they are also used for some security functionalities, such as using the heart rate sensor, to ensure the owner who unlocked the watch, has not removed the watch since authorization occurred.

2.1.1 Heart rate sensor

The integrated heart rate sensor uses the technology *photoplethysmography*. Which functions by using green LED's to illuminate the veins in the wrist and photodiodes to detect the amount of blood flowing in said veins. This allows the watch to detect the wearers heart rate [8]. The quality of measurements rely on the wearers fit of the watch, but when fitted correctly heart rate data is provided through Apple's *HealthKit* framework. The heart rate data can be fetched through an instance of `HKHealthStore`, and can both be fetched as a stream through a `WorkoutSession` and retroactively through queries to the `HKHealthStore` [5]. Heart rate data for biometric identification has been explored by other vendors than Apple [14], but this has been done with different sensors and raw sensor data. It has been found possible with raw sensor data to identify individuals from photoplethysmography signals, like the ones obtained from the Apple Watch sensor [15]. A photoplethysmography sensor therefore seems like a potential biometric characteristic capture device.

2.1.2 Accelerometer & Gyroscope

Both an accelerometer and a gyroscope are located in the watch, and these are used for features like activity tracking and rotation detection for screen auto on/off. Raw data is accessible from both sensors through the *CoreMotion* framework, from `CMAccelerometerData` and `CMGyroData` accordingly [1]. The accelerometer allows for 3-axis movement tracking, and the gyroscope detects rotation. These in combination allow for precise movement tracking, but are limited to track the movement of the wrist. This has been used for activity tracking and pedometer in many smart devices, but one might need to investigate the precision of this before using it for identification. Experiments on identifying individuals

from motion data has been tested, and one approach could be gait recognition. Gait recognition allows for identification from how the subject walks, and research points towards this being possible from similar sensors within smartphones [11].

2.1.3 Microphone

The watch contains a microphone, used for voice commands and speakerphone. The microphone can be accessed and used for recordings by the usage `AudioRecorderController` spawned with the function `presentAudioRecorderControllerWithOutputUrl`.

2.1.4 NFC

The potential of an NFC chip in the watch seems promising for access control, as it could allow for interaction with third-party devices, i.e. NFC readers. This could be utilized for doors which rely on NFC key cards, where the watch could function as a secure device containing the NFC key cards. Unfortunately this is simply not possible, as Apple has restricted developer access to the NFC chip, only allowing it to be used with their own *Apple Pay* service.

2.2 Limitations

The SDK does not allow for full control of all aspects of the watch. There are some limitations to the way Apple runs third-party applications on the platform. Most of the limitations are of course present due to user experience and battery lifetime constraints. The watch does not allow for continuous background services, it is dependent on having an iPhone connected to enable internet connectivity, apps can not hinder the screen in automatically shutting off and the processing power is of course limited in comparison to smartphones and laptops.

The main limitations affecting the development of the prototype are the lacking ability to run background services and the automatic screen on/off.

The ability to run background services would be necessary if something like gait recognition should function. This is not possible, and limits the continuous tracking of an individual for identification purposes.

If identification should rely on movement data from the accelerometer and gyroscope, the automatic screen control is a limiting functionality. When one moves the watch the screen very likely turns off, and the process halts. It is possible to start workout sessions (*HKWorkoutSession*) [6] and extract the data from here, so a workaround might be possible, in order to extract continuous data.

2.3 System Proposals

With the sensors and limitations taken into account, three systems will be proposed. These systems will be evaluated on their possibility of implementation, and their usefulness. The proposed systems will form the basis for the prototype.

2.3.1 Voice Recognition

This approach could utilize the built-in microphone to record and extract features from the subjects voice. The detection could rely both on the words being said and not. Voice recognition in general has seen an immense advancements, due to the growing popularity and efficiency of modern machine learning technologies [13]. This approach would therefore be both possible and convenient for users, already comfortable with talking to their smart devices. The system would need to extract features of the subjects voice, not only said words. This is quite challenging and might also need machine learning techniques to perform adequately. Due to the limited processing power of the Apple Watch, and the requirements from machine learning, the recorded sound samples might need to be sent to a central server for processing. This could result in privacy issues, as biometric authentication data is sent over the network. It could also result in significant latency, worsening the user experience.

2.3.2 Gait Recognition

Using the accelerometer and gyroscope within the watch, the continuous movement data could be used to identify the individual wearing the watch, i.e. gait recognition. This approach could be very convenient, as it does not require any interaction from the wearer besides wearing it, making it very pervasive. The main problem with this, is the continuous measurements from the Apple Watch, and the limited processing power of the watch [12]. Since the Apple Watch does not support background services on the watch, the continuous data retrieval might not be possible. Furthermore gait recognition from smartwatch movement data, has the potential of being a research project in itself. Research has been performed on how to achieve this with smartphones [12], but the phone is normally located in pocket, whereas the watch is on the wrist, resulting in different movement data. As with voice recognition, gait recognition requires machine learning techniques such as *principal component analysis* to identify patterns and characteristics in the data, which would require the movement data to be sent to central servers, as the Apple Watch is neither powerful or efficient enough to perform these kinds of calculations [10].

2.3.3 Photoplethysmography Recognition

As mentioned the built-in heart rate sensor utilizes the technology *photoplethysmography (PPG)* to detect the wearers heart rate. It is shown the raw data from sensors like these can

be used for identification [15]. This approach like the others, require access to raw data, but does not require it continuously as with gait recognition. Unfortunately the recognition of subjects again requires advanced processing, limiting the viability of such an approach. Another problem is the access to raw PPG data from the watch. Apple only allows access to preprocessed data, i.e. the calculated heart rate.

3 Prototype

Due to the findings in the analysis section, the prototypes scope has been limited to feature extraction from the different sensors in the watch. This means that no feature comparison will be performed. This was found to be a necessary compromise, as all the proposed systems requires advanced machine learning to function optimally. An implementation of the *Dynamic Time Warp* algorithm has also been included, to test if any comparison algorithm is possible on a smart watch [9]. Much of the code has been inspired by open source examples [4] [3] [7], found on the popular code repository site *GitHub*. The main purpose of the prototype is therefore to sample the sensors, allowing for further analysis of the output.

The prototype includes feature extraction from the three possible sensors, i.e. the microphone, accelerometer and heart rate sensor. These should be easily sampled, and extracted from the device, this process will also be covered.

3.1 User Interface

The developed user interface found in figure 1, allows the wearer to perform measurements with the watch. This includes samples of heart rate, voice and movement. Testing of the *Dynamic Time Warping* has also been included, which involves running *DTW* on a two arrays of length 1000. The different samplings can be configured upon spawning of their views, this is done from the main views controller, `MainInterfaceController`, see listing 1.

List. 1: Spawning of ViewControllers from the main interface, passing contexts setting up the sampling.

```
@IBAction func movementButtonTapped() {
    let context = MovementInterfaceContext()
    context.instruction = "Sampling Movement"
    context.dataStorePath = "Movement_sample_\(NSTimeIntervalSince1970).data"
    context.sampleDuration = 10.0
    context.completionClosure = {() -> Void in print("Done")}
    self.pushControllerWithName("movementScene", context: context)
}

@IBAction func voiceButtonTapped() {
    let context = VoiceInterfaceContext()
    context.instruction = "Sampling Voice"
```

```
    context.dataStorePath = "Voice_sample_\(NSTimeIntervalSince1970).mp4"
    context.sampleDuration = 10.0
    context.completionClosure = {() -> Void in print("Done")}
    self.pushControllerWithName("voiceScene", context: context)
}

@IBAction func heartRateButtonTapped() {
    let context = HeartRateInterfaceContext()
    context.instruction = "Sampling Heart"
    context.dataStorePath = "HeartRate_sample_\(NSTimeIntervalSince1970).data"
    context.sampleDuration = 100.0
    context.completionClosure = {() -> Void in print("Done")}
    self.pushControllerWithName("heartRateScene", context: context)
}

@IBAction func dtwTestButtonTapped() {
    let context = DynamicTimeWarpInterfaceContext()
    context.instruction = "Testing DTW"
    context.testSampleSize = 1000
    self.pushControllerWithName("dtwScene", context: context)
}
```

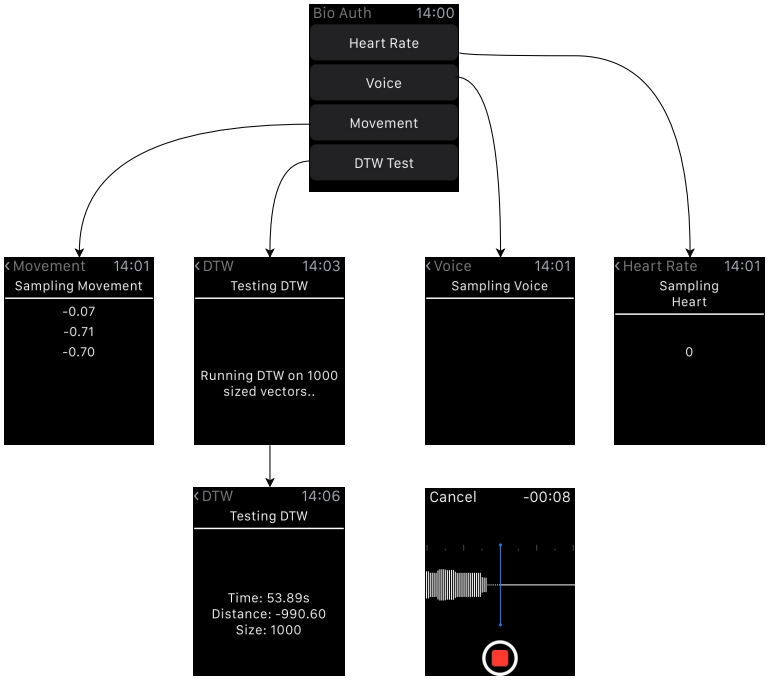

Fig. 1: Flow of the developed prototypes user interface. Start menu allows for data extraction from the three sensors and to run DTW on arrays with 1000 entries.

3.2 Microphone

All functionality related to the microphone is handled by the microphone view, which is controlled by the `VoiceInterfaceController`. Here the recording interface is spawned to record the specified time. The recording can be performed in a variety of qualities and formats, and is spawned with the `WKInterfaceController` function `presentAudioControllerWithOutputURL`, see listing 2. The format is controlled by the file extension on the `NSURL` which the function saves the recording to, and the quality is defined in the `preset`. The different presets can be found in table 2.

List. 2: Spawning of the audio recording controller.

```
self.presentAudioRecorderControllerWithOutputURL(
    recFileUrl,
    preset: WKAudioRecorderPreset.WideBandSpeech,
    options: [
        WKAudioRecorderControllerOptionsMaximumDurationKey:
            NSTimeInterval(duration)
    ],
    completion: { (didSave, error) -> Void in
        print("error: \(error)\n")
        if didSave {
            self.storeVoiceData(toFile, dataUrl: recFileUrl)
            self.popController()
        } else { self.popController() }
})
```

Tab. 1: Available audio recording presets, their sampling rate and bit rate.

Audio Recorder Presets		
Preset	Sample Rate	Format / Bit Rate
HighQualityAudio	44.1 kHz	LPCM 705.6 kbps or AAC 96 kbps
WideBandSpeach	16 kHz	LPCM 256 kbps or AAC 32 kbps
NarrowBandSpeach	8 kHz	LPCM 128 kbps or AAC 24 kbps

3.3 Accelerometer

The accelerometer is accessed through the `CoreMotion` framework, which provides a `CMMotionManager` one can attach sensor handlers to. Through the manager it is possible to check if the accelerometer is available, set its update interval and attach a handler of the class `CMAccelerometerHandler`. The handler receives updates of the defined interval which, in the prototype, is saved to arrays for later storage. See listing 3 for an example of how the accelerometer is accessed in the prototype. On the iPhone this is also the approach to retrieving data from the gyroscope, but the watch returns `false` on the managers function `gyroActive`.

List. 3: Usage of accelerometer in prototype. The accelerometerHandler is attached to the motionManager in order to receive updates on the set interval of 0.1.

```
motionManager.accelerometerUpdateInterval = 0.1
if motionManager.accelerometerAvailable {
    let accelerometerHandler:CMAccelerometerHandler = {
        (data: CMAccelerometerData?, error:NSError?) -> Void in
        if let data = data {
            self.xLabel.setText(String(format: "%.2f", data.acceleration.x))
            self.yLabel.setText(String(format: "%.2f", data.acceleration.y))
            self.zLabel.setText(String(format: "%.2f", data.acceleration.z))
            self.measuredDataX.append(data.acceleration.x)
            self.measuredDataY.append(data.acceleration.y)
            self.measuredDataZ.append(data.acceleration.z)
        }
        if let error = error {
            print(error.localizedDescription)
        }

    }
    if let duration = localContext?.sampleDuration {

        motionManager.startAccelerometerUpdatesToQueue(
            NSOperationQueue.currentQueue()!,
            withHandler: accelerometerHandler)

        dispatch_after(dispatch_time(DISPATCH_TIME_NOW,
                                     Int64(duration * Double(NSEC_PER_SEC))),
                                     dispatch_get_main_queue()) {

            self.motionManager.stopAccelerometerUpdates()
            if let location = self.localContext?.dataStorePath {
                self.storeAccelerometerData(location,
                                            dataX: self.measuredDataX,
                                            dataY: self.measuredDataY,
                                            dataZ: self.measuredDataZ)
            }
        }
    }
}
```

3.4 Heart Rate Sensor

Heart rate data is accessed through the `HealthKit` framework. As mentioned it is only possible to extract heart rate, and not the raw PPG data. The heart rate sensor is started by starting a `HKWorkoutSession`, which then records the health statistics, including the heart rate. The session is started and stopped through the `HKHealthStore`, which also handles the following queries for heart rate data. A query to the health store lets the caller define sorting, limits to number of returned samples and predicates, such as start and end time of the queried measurements, see listing 4. The result from the query is

injected into the `resultsHandler`, which in the prototype sends the data to the phone. In order to access the health data, one needs explicit permission by the owner of the device, which is requested on the watch and handled on the phone, through the health stores function `requestToShareTypes`. Beyond allowing for queries to the health store, after ended workout session, it is also possible to receive a continuous stream of heart rate data, through a `HKAnchoredObjectQuery`. This is utilized in the prototype to update the heart rate label on the watch, see listing 5. A sample time of 100 seconds, results in approximately 20 samples of the heart rate. The heart rate is not very useful for identification, but one could hope that future versions will allow for raw data extraction.

List. 4: Setting up a query for heart rate samples. Including a time sorting descriptor, a date predicate and a handler for the returned results.

```
let sortByTime = NSSortDescriptor(key: HKSampleSortIdentifierEndDate,
                                  ascending: false)
let datePredicate =
    HKQuery.predicateForSamplesWithStartDate(startDate,
                                             endDate: endDate,
                                             options: HKQueryOptions.None)
let query =
    HKSampleQuery(
      sampleType: heartRateType,
      predicate: datePredicate,
      limit: 1000,
      sortDescriptors: [sortByTime],
      resultsHandler: {(query, results, error) in
        guard let results = results else { return }
        for sample in results {
            let quantity = (sample as! HKQuantitySample).quantity
            let heartRateUnit = HKUnit(fromString: "count/min")
            heartRateArray.append(quantity.doubleValueForUnit(heartRateUnit))
        }
        ...
    })
self.healthStore.executeQuery(query)
```

List. 5: Setting up a streaming heart rate query for continuous handling of samples generated by the heart rate sensor.

```
func getHeartRateStreamingQuery(workoutStartDate: NSDate) -> HKQuery? {
    guard let quantityType = HKObjectType.quantityTypeForIdentifier(
                                HKQuantityTypeIdentifierHeartRate)
                            else { return nil }
    let heartRateQuery =
        HKAnchoredObjectQuery(
            type: quantityType,
            predicate: nil,
            anchor: anchor,
            limit: Int(HKObjectQueryNoLimit))
            {(query, sampleObjs, deletedObjs, newAnchor, error) -> Void in
```

```
                guard let newAnchor = newAnchor else { return }
                self.anchor = newAnchor }

    heartRateQuery.updateHandler =
        {(query, sampleObjs, deletedObjs, newAnchor, error) -> Void in
            self.anchor = newAnchor!
            self.updateHeartRate(sampleObjs)}
    return heartRateQuery
}
```

3.5 Dynamic Time Warp

The implementation of DTW is only included to showcase the computational power of the *Apple Watch*, and to evaluate if it is viable to run such a rather heavy algorithm on the watch. The DTW algorithm has beenMotion Sensors implemented in the class `DynamicTimeWarper`. The class only has a single function, `distance`, which takes to arrays of doubles, and returns the calculated distance between the two. The algorithm is run by the `DynamicTimeWarpInterfaceController`, which tracks how long time the algorithm takes to execute the algorithm, on arrays filled with random data. As seen in table **??** the complexity is as expected $O(n^2)$. This makes it viable to run smaller sets with this algorithm, and other distance measuring algorithms could probably also run on the device. Another approach could be to execute the algorithm on the connected iPhone, which would result in a significant performance boost, due to the more powerful hardware. This would allow for more complex algorithms on largers sets of data.

List. 6: Testing of the implemented DTW algorithm, within the DynamicTimeWarpInterfaceController

```
dispatch_async(queue) {
    let dtw = DynamicTimeWarper()
    let testArr1 =
        (0...localContext.testSampleSize!).map {_ in drand48()}
    let testArr2 =
        (0...localContext.testSampleSize!).map {_ in drand48()}
    let distance =
        String(format: "%.2f", dtw.distance(testArr1, target: testArr2))
    let deltaTime =
        String(format: "%.2f", abs(self.dtwStart!.timeIntervalSinceNow))
    self.resultLabel.setText("Time: \(deltaTime)s\n
                              Distance: \(distance)\n
                              Size: \(localContext.testSampleSize!)")
}
```

Tab. 2: Measured time it takes to run DTW on different sized arrays.

Array Size	Time in seconds
100	
250	
500	
1000	53.89

3.6 Data transfer and storage

In order to extract the data from the watch, it is sent to the iPhone in a message. The iPhone ensures that the data is stored in the *Documents* directory, so the data easily can be extracted. This process happens after each sampling from a sensor. The library `WatchConnectivity` handles transfers of messages and files through sessions. The session is initiated both on the watch, and on the phone. An example of usage on the watch can be found in listing 7 and the receiving example can be found in listing 8. After succesful transfer, the samples can be downloaded from the phone through iTunes. This approach could also be used if processing of the measured data should be performed on the phone instead.

List. 7: Send heart rate data to the phone from the watch.

```
self.session = WCSession.defaultSession()
self.session!.sendMessage(
    ["heartRateArray": heartRateArray, "location": location],
    replyHandler: {(response) -> Void in
        print("Succesful heart-rate msg to phone")
    },
    errorHandler: {(error) -> Void in
        print("Unsuccesful heart-rate msg to phone")
    })
```

List. 8: Receiving heart rate data on phone, and storing it to the Documents folder.

```
func session(session: WCSession,
             didReceiveMessage message: [String : AnyObject],
             replyHandler: ([String : AnyObject]) -> Void) {
...
    let documentsDir = try NSFileManager.defaultManager()
                           .URLForDirectory(.DocumentDirectory,
                                            inDomain: .UserDomainMask,
                                            appropriateForURL: nil,
                                            create: true)
    guard let location = message["location"] as? String else {return}
    if let heartRateArray = message["heartRateArray"] as? Array<Double> {
        if let pathUrl = NSURL(string: location, relativeToURL: documentsDir) {
            try heartRateArray.description
                              .writeToURL(pathUrl,
                                          atomically: true,
                                          encoding: NSASCIIStringEncoding)
```

```
                replyHandler([:])
            }
        }
...
```

References

[1] Core motion framework reference, 2013. https://developer.apple.com/library/ios/documentation/CoreMotion /Reference/CoreMotion_Reference/, Apple Inc., Fetched: 15.06.2016.

[2] Apple watch - technical specifications, 2016. https://support.apple.com/kb/SP735?locale=en_US, Apple Inc., Fetched: 12.06.2016.

[3] Bradlarson/healthkitheartrateexporter, 2016. https://github.com/BradLarson/HealthKitHeartRateExporter, GitHub, Fetched: 16.06.2016.

[4] coolioxlr/watchos-2-heartrate, 2016. https://github.com/coolioxlr/watchOS-2-heartrate, GitHub, Fetched: 16.06.2016.

[5] The healthkit framework, 2016. https://developer.apple.com/library/ios/documentation /HealthKit/Reference/HealthKit_Framework/, Apple Inc., Fetched: 15.06.2016.

[6] Hkworkoutsession class reference, 2016. https://developer.apple.com/library/watchos/documentation/HealthKit /Reference/HKWorkoutSession_ClassReference/index.html.

[7] shu223/watchos-2-sampler, 2016. https://github.com/shu223/watchOS-2-Sampler, GitHub, Fetched: 16.06.2016.

[8] Your heart rate. what it means, and where on apple watch youâll find it., 2016. https://support.apple.com/enus/HT204666, Apple Inc., Fetched: 13.06.2016.

[9] BERNDT, D. J., AND CLIFFORD, J. Using dynamic time warping to find patterns in time series. In *KDD workshop* (1994), vol. 10, Seattle, WA, pp. 359–370.

[10] DAWSON, M. R. Gait recognition.

[11] FERRERO, R., GANDINO, F., MONTRUCCHIO, B., REBAUDENGO, M., VELASCO, A., AND BENKHELIFA, I. On gait recognition with smartphone accelerometer. In *2015 4th Mediterranean Conference on Embedded Computing (MECO)* (June 2015), pp. 368–373.

[12] FERRERO, R., GANDINO, F., MONTRUCCHIO, B., REBAUDENGO, M., VELASCO, A., AND BENKHELIFA, I. On gait recognition with smartphone accelerometer. 368–373.

[13] HINTON, G., DENG, L., YU, D., DAHL, G. E., R. MOHAMED, A., JAITLY, N., SENIOR, A., VANHOUCKE, V., NGUYEN, P., SAINATH, T. N., AND KINGSBURY, B. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine 29*, 6 (Nov 2012), 82–97.

[14] INC., N. Heartid white paper.

[15] KAVSAOGLU, A. R., POLAT, K., BOZKURT, M. R., AND MUTHUSAMY, H. Feature extraction for biometric recognition with photoplethysmography signals. *Signal Processing and Communications Applications Conference* (2013).