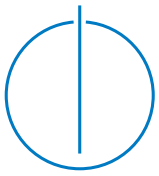
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Pattern Recognition with Smart Devices as Personal Authentication Factor

Philipp Fent



DEPARTMENT OF INFORMATICS

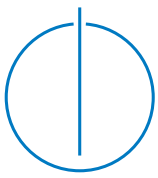
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Pattern Recognition with Smart Devices as Personal Authentication Factor

Mustererkennung mit Mobilgeräten als persönliches Identifikationsmerkmal

Author:	Philipp Fent
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Nils T. Kannengießer, M.Sc, ~~Prof. Senjun Song, Ph.D.~~ Prof. Dr. Sejun Song modified by NK 22.04.2016
Submission Date:	15. February 2016



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15. February 2016 Philipp Fent

Acknowledgments

Special thanks go to Prof. Dr. Senjun Song and Nils Kannengießer, who gave me the opportunity to work on an international project. Nils especially guided and taught me many things used in this thesis, especially during the Android Practical Course in summer 2015.

Many thanks also go to the Chair of Operating Systems at Technische Universität München (TUM) for hosting this thesis and Prof. Dr. Uwe Baumgarten as a supervisor. The Chair of Operating Systems provided test devices, crucial for the concept of this thesis.

I also like to thank several students from the Andoid practical course 2015/16 for generation of test data and H. Wang, T. Lai and R. Choudhury for their paper on Motion Leaks[36], which initially sparked the ideas outlined in this thesis. Also thanks to many unnamed proofreaders, reviewers and everyone that supported me during writing.

Abstract

Every person displays characteristic patterns of behavior, that can be used to verify her or his identity. With the rise of personal smart devices, e.g. smartwatches or smartphones, these patterns can be recorded and analyzed. The resulting characteristics can be used as an additional factor in Multi-Factor Authentication or used as an intrusion detection system by reporting anomalies. In this thesis, we analyze the patterns for motion, determined by measuring acceleration. We then evaluate how to efficiently, accurately, and practically extract behavioral patterns, identifying individual users. Furthermore, we developed Android smartphone and smartwatch app prototypes demonstrating the identification capabilities.

Menschen besitzen charakteristische Verhaltensmuster, durch die sie eindeutig identifiziert werden können. Mit ständig mitgeführten Mobilgeräten wie Smartphones oder Smartwatches, können diese Muster aufgezeichnet und analysiert werden. Die daraus abgeleiteten Merkmale können als zusätzlicher Faktor bei Multi-Faktor-Authentifizierungsverfahren oder als Angriffserkennungssystem eingesetzt werden. Im Rahmen dieser Arbeit werden diese Muster anhand von Beschleunigungssensoren hinsichtlich effizienter und präziser Verhaltensmustererkennung analysiert. Diese Mustererkennungsverfahren werden durch Prototypen einer Android Smartphone und Smartwatch App demonstriert.

Contents

1 Introduction

Modern computer systems are facing various threats of being attacked. Users are one of the most commonly exploited gateways to computers, especially with social engineering and fishing. Most systems currently rely on a standard combination of username and password. In 2009, Aloul et al. described the most common security concerns with passwords: “Users tend to use easy-to-guess passwords, use the same password in multiple accounts, write the passwords or store them on their machines, etc. Furthermore, hackers have the option of using many techniques to steal passwords such as shoulder surfing, snooping, sniffing, guessing, etc.” [4] .

Additionally, many users are constantly logged in to services with their mobile devices, despite not having appropriate security measures for their devices. On most Android phones, full disk encryption is not enabled by default [33], which leads to another attack vector for identity theft.

Aloul et al. introduced a system of One Time Passwords (OTPs) using mobile phones, which significantly improves security by introducing a second factor of authentication. However, for authentication situations on smartphones themselves the OTP mechanism is rendered pretty much useless, as the second factor is in fact on the same device.

1.1 Vision

The vision of this thesis is to provide a way to easily detect individual users by a short authentication sequence based on acceleration patterns. Even though this does not necessarily qualify as cryptographically secure authentication, behavioural patterns and keystroke recognition can be used as biometric authentication aids. For example, Bhargav-Spantzel et al. [7] described a system to extract cryptographic biometric keys from biometric data and showed how this can be combined with additional other proofs of identity to provide strong authentication.

With acceleration pattern recognition, we are able to create an authentication mechanism with zero additional user interaction. This allows higher frequency user re-authentication without disturbing or annoying the user. That means, instead of prompting the user with a login screen every 24 hours, we can measure his or her acceleration patterns every single time sensitive information is accessed. Therefore we can not only provide basic login authentication, but also provide a way for users to stay authenticated

for longer sessions or even detect when someone else hijacks a valid session. Since the time between two authentication requests can be arbitrarily small, an attacker who gets control over the current session will be deauthenticated quickly, thus minimizing the potential damage.

1.2 Single user licenses

Another application of this technique is to identify individual users, even when using the same device. This might not only be useful in terms of individualizing the software according to the current user, but also for monitoring software usage.

A common licensing model for software are per-user licenses, i.e. n licenses for n users of the software. However, this license model currently can not be enforced since software is installed on a single physical device, which may be shared among users. This led to most software companies licensing their software per-installation instead of per-user. As of 2016, many users tend to have multiple devices and also want to use their licenses on all of them. This resulted in a trend to bind software licenses to user accounts instead of devices. The trend is also prevailing in modern Software as a service (SaaS) models, which do not require installation of the software on end-user devices anymore. A possible circumvention of these account bound licenses is account sharing. This imposes a real problem, not only for software licensors, but also for other access providers. For example a consumer research from Parks Associates reports, that “6% [of video streaming users] are exclusively using shared accounts to access subscription” [29].

The methods described in this thesis provide a powerful way to detect individual users, sharing physical devices or accounts. Thus with this approach we might reduce copyright infringements that could not be detected beforehand.

1.3 Multi-factor authentication

Multi-factor authentication (MFA) is a technique to enhance security in access control situations. It combines multiple forms of authentication mechanisms, based on conceptually different approaches: Knowledge, e.g. passwords or PINs; possessions, e.g. keys or bank cards; and biometric characteristics, like fingerprints or behavioural patterns.

A typical authentication attempt with MFA is only successful, when all needed factors are present. The most common example for MFA is banking, where one individual needs to be in possession of the banking card and also needs to know the card’s PIN. However, an attack vector targeting this system is copying the banking card while the attacked

person does not notice, that his card is being attacked. This attack vector is also possible with biometric characteristics and even relatively easy, as many biometric traits are publicly visible. Fingerprints have proven to be copyable with low cost [15] and new high resolution cameras allow to photograph fingerprints and eyes in high enough quality to spoof many scanners [16]. These attacks can be adapted to other authentication systems based on visible biometric traits, such as iris recognition or Android's Face Unlock.

For biometric authentication to be sufficiently secure, the traits need to be intrinsic, i.e. not publicly visible and hard to copy. Acceleration based motion detection matches these requirements, as recording of these patterns is only possible with physical access to the authentication device or extremely precise monitoring of all body movement of the user.

1.4 Smart mobile devices

Smart devices are electronic devices, that feature wireless communication, e.g. WiFi or Bluetooth. Smart *mobile* devices are smart devices, that are typically worn or kept in close proximity to the user. This usage usually results in small form factors and little weight. These devices are most often commodity devices and used frequently. Therefore, smart mobile devices are ideal to provide authentication, since the authenticating user is accustomed using the device.

Currently common examples for smart mobile devices are smartphones. With the release of the Apple Watch last year in 2015, smartwatches became increasingly popular. With a rapid miniaturization of smart mobile devices, a consequent next step would be even smaller devices, like smart-rings. In Figure 1.1 a size comparison between the three mentioned categories is shown.

One common trait of almost all smart mobile devices is the presence of acceleration and gyroscope sensors. These sensors are reasonable small and available as integrated circuits to fit in even the smallest devices. For this thesis, we are focusing on acceleration sensors to detect behavioural patterns.

1.4.1 Smartphones

Smartphones are the most capable of the smart mobile devices discussed herein. Smartphones usually have numerous wireless communication possibilities and thus function are a personal data-hub, to which other personal devices connect and communicate over. Typical connections for Smartphones are: Cellular network (e.g. GSM, UMTS), WiFi, Bluetooth and Near Field Communication. Smartphones are also packed

Figure 1.1: Examples for smart mobile devices that are worn in close proximity of the user (from left to right): A OnePlus One smartphone, a Sony SmartWatch 3, Smarty Ring concept design [1, 2, 3]

with sensors, which can be utilized in apps and typically include acceleration as well as gyroscopic sensors for movement detection.

The three major operating systems of smartphones are Android, iOS and Windows. As shown in Figure 1.2, Android has the biggest market share of over 80% and an app targeting both Android as well as iOS can reach about 98% of the smartphone market.

1.4.2 Smartwatches

Smartwatches, as displayed in the middle of Figure 1.1, are a newer iteration of wearable smart mobile devices. Early designs of smartwatches (essentially calculators) came up in the 1980's and first prototypes were released in the 1990's. However, modern smartwatches feature network connectivity and are usually paired with smartphones via Bluetooth.

Currently, the biggest market share, as shown in Figure 1.3, with over 50% in smartwatches has Apple with the Apple Watch, running watchOS. Android Wear, an adapted version of the Android operating system for smartwatches, has the second highest market share. All in all, the smartwatch market is more diverse than the smartphone market, with several independently developed operating systems like Pebble OS or

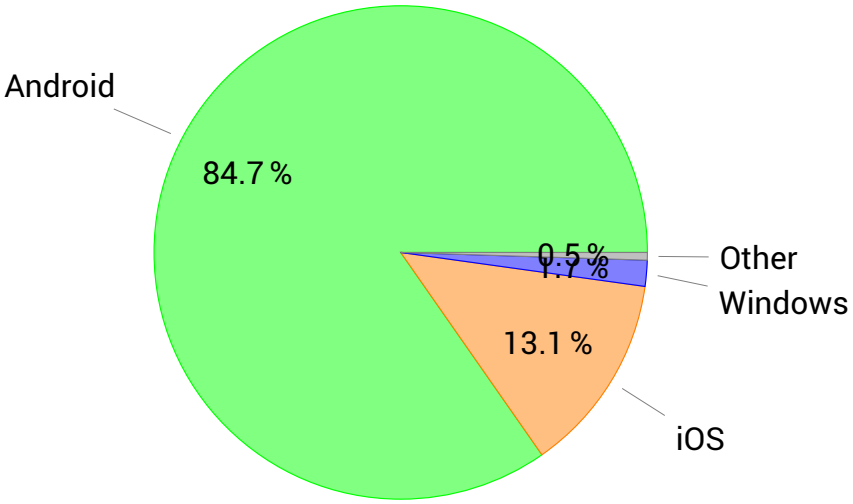


Figure 1.2: Market share of smartphone operating systems in Q3'15 [18]

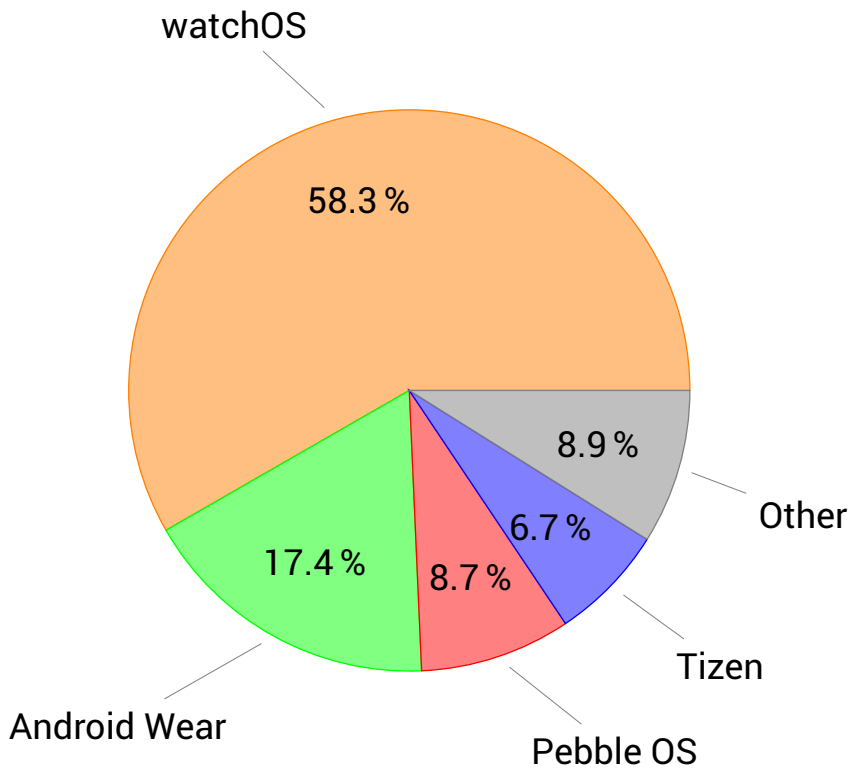


Figure 1.3: Market share of wristware operating systems in 2015 [19]

Samsung's Tizen. Nonetheless, Android Wear has the advantage of having consistent Application Programming Interfaces (APIs) with Android.

With developing apps for Android in combination with small adaptions to Android Wear, developers can target a huge percentage of smartphones and also develop smartwatch apps with virtually no overhead.

1.4.3 Smart-rings

Smart-rings are the next step towards even smaller wearables. The concept of these rings is to provide the same basic "smart" functionality comparable to a smartwatch, without the need to actually wear a watch. For example displaying notifications or providing authentication for payment processes can also be done solely on a smart ring.

However, there are no commercially available smart-rings yet, but only design concepts and prototypes. Specially developed smart-rings with acceleration sensors should be able to identify users based on their motion patterns. Future work might provide basic functionality on smart-rings, but definitely will need adaption to special purpose operating systems.

1.5 Pattern recognition

Pattern recognition is a branch of machine learning, that is focused on determining the similarity of data sets. This is usually used to detect predefined patterns, e.g. reading numbers or detecting gestures. However, this approach can also be used to generally group similar data sets and classify these without predefined categories. Grouping acceleration patterns is the main goal of this thesis, so pattern recognition algorithms are a main focus.

The pattern recognition concepts and techniques are largely based on Bishop's book "Pattern Recognition and Machine Learning" [8]. We are especially using the concept of a three staged data processing, as shown in Figure 1.4. In this concept, we are first record the raw data, then preprocess it and extract a set of predefined features before using a standard classification algorithm.

1.5.1 Preprocessing

In theory, pattern recognition algorithms can work on raw data. However, preprocessing steps can significantly improve the results of these algorithms. In practical applications, preprocessing is used to transform the data, so that the complexity of the problem is reduced.

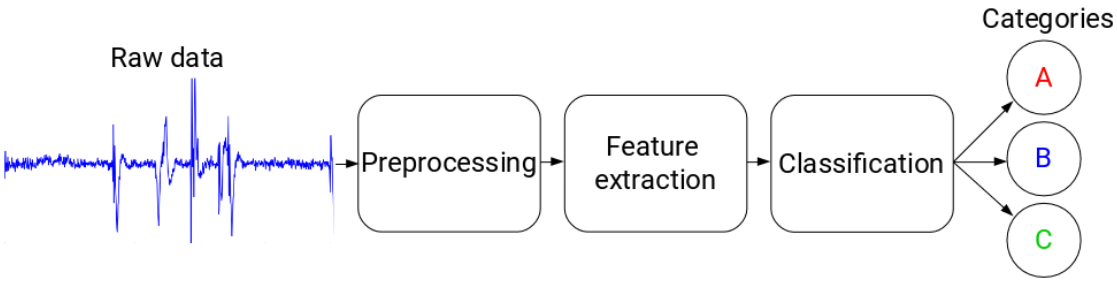


Figure 1.4: The three staged pattern recognition approach, implemented in this thesis

A common example using this technique is recognising digits in images. The digit recognition get significantly easier, when the image is cropped to a square around the digit and reduced to a black and white image. Generally speaking, preprocessing steps usually scale the problem to a fixed size and eliminate random noise. As a result, all data is transformed to have similar shape and in general gains predictability.

The preprocessing steps usually operate on the raw input data and do not result in a loss of information.

1.5.2 Feature extraction

The next step, feature extraction, uses more aggressive means to extract the relevant data for our specific use-case. The feature extraction step usually reduces the data set by several orders of magnitude, which makes subsequent categorization much faster and increases reliability. As a note, we are making a distinction between preprocessing and feature extraction, whereas Bishop [8] treats them as the same.

Feature extraction is a core component of this thesis, since the performance of the pattern recognition algorithms scales with the amount of data needed to process. The main idea of this step is, that we do not need to compare the whole data from the acceleration sensors, but only “features” we identify in the feature extraction step.

However, this feature extraction can also result in information loss. This means, that a precise and accurate feature extraction algorithm is essential. A too broad feature extraction algorithm can result in not recognizing legitimate authentication attempts, while a too narrow algorithm allows attackers to easily bypass this system. Also, for the data to be comparable, all data must undergo the same preprocessing and feature extracting steps. Even slight changes in the algorithms can spoil previous training steps.

1.5.3 Classification

In the end, a pattern recognition algorithm is used to classify the data into various categories. This is usually done by training the algorithm with known categories first. Then, based on this training, the algorithm can decide to which category new data, with a previously unknown category, belongs to. This form of training is called “supervised” training.

In general, the training can also be done unsupervised, i.e. the initial categories are unknown and the algorithm finds best-fit categories itself. However for this thesis, we implemented a supervised algorithm, which only identifies previously learned users. Automatically recognising new users might be possible as a future enhancement.

2 Approaches

2.1 Key stroke pattern recognition

Timings and patterns in key strokes are individual characteristics, that can serve as biometric user identification. Individual typing patterns can be extracted from typing samples and later be used to verify a users identity. The patterns, so called keystroke dynamics, are usually extracted via the key down / up events. Dholi and Chaudhari [12] classified several features from those events, as shown in Figure 2.1: The interval between two key presses, the dwell time of a single key press, the latency between consecutive keystrokes, the flight time and the time from one up event to another.

2.1.1 Related work

The idea of authentication by keystroke timings started as early as 1980 with Gaines et al. [17] to evaluate the effectiveness experimentally. In their experiment, the scientists gave seven professional typists, i.e. secretaries, a text to type and examined their patterns in typing. Gaines et al. looked at the time to type pairs of successively typed letters, so called “digraphs”. From five of these digraph timings, all of the seven typists in this study could be identified.

In 1997, Dieter Bartmann presented PSYLOCK [6], a system that analyzed the keystroke rhythm of text input and identified users according to these rhythms. The system claimed to work with an arbitrary text of aprox. 100 characters. PSYLOCK used an approach based on statistical models in combination with support vector machines.

In 1999, Monrose and Rubin [27] proposed a new approach at keystroke identification: Continuous keystroke verification. Previous attempts to keystroke verification only used static verification, i.e. they verified the characteristics only at specific times, for example during login. Monrose’s and Rubin’s system monitored the user’s typing behaviour continuously and thus could provide a significant improvement of security. The basic approach in their classification algorithm was clustering feature sets, determined through factor analysis, with a k-Nearest Neighbors (k-NN) approach.

Clarke and Furnell [10] published a paper on identifying mobile phone users using keystroke analysis in 2006. In this paper, users were identified by typical handset interactions: entering telephone numbers and writing text messages. Clarke and Furnell also

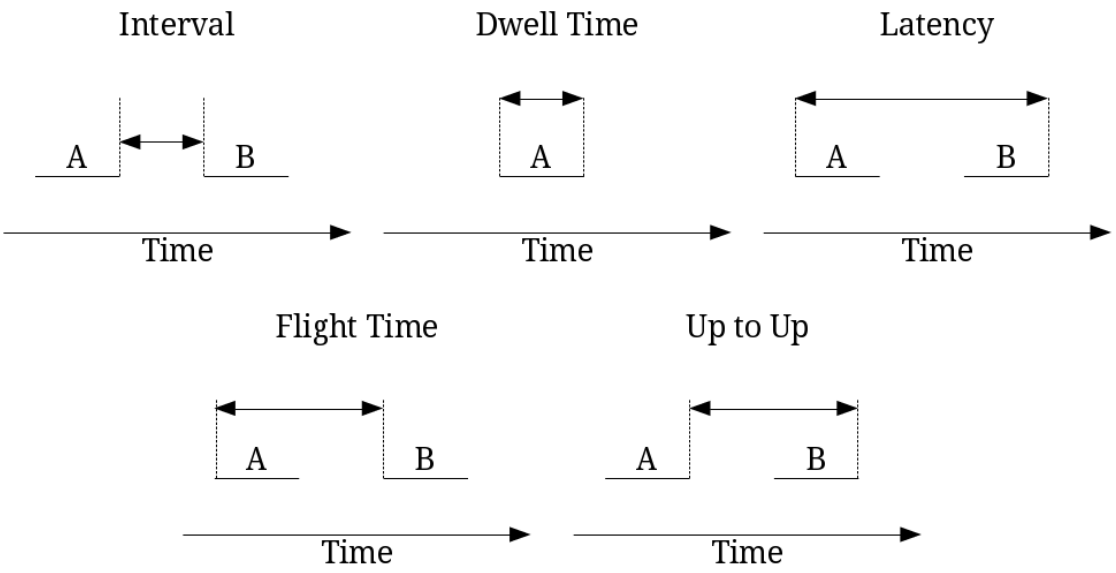


Figure 2.1: The five temporal features of keystroke dynamics [12]

compared several multi-layered neuronal networks of different flavour, all of which performed approximately the same. Since the form factor of mobile phones has drastically evolved in the last 10 years, their approach is largely obsolete for modern smartphones lacking physical buttons.

2.1.2 Identification of keystrokes based on acceleration data

Since smartphones lack physical buttons, but contain several high accuracy motion sensors, correlating taps on the screen with keystrokes seems not far fetched. Typical character input on phones is done via on-screen keyboards controlled via taps on the screen.

In the approach of this thesis, identifying keystroke patterns with acceleration sensor data, we are not limited to simple character and text input, but can also identify arbitrary tap sequences, that might occur in authentication scenarios. For example reading emails or messages is equally or even more secure-worthy than writing messages. Our approach can also extract and match patterns to authenticate users in simple navigation flow.

As a side note, typically the access to the on-screen keyboard is significantly restricted for apps to hinder keyloggers from sniffing passwords. However, access to acceleration sensors is pretty much unrestricted and can also be done in web-browsers such as Google Chrome via an Javascript API [9].

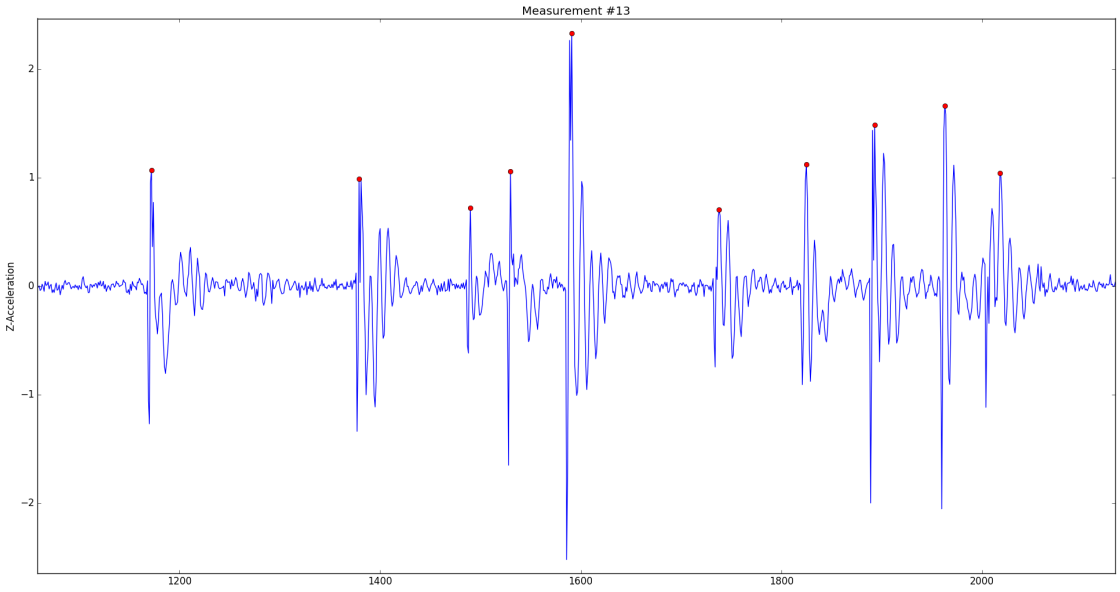


Figure 2.2: Detected peaks in sensor measurements (marked in red), as described in Section 3.2.4

In 2012, Miluzzo et al. [26] introduced *TapPrints*, a mechanism to extract the location of screen taps solely from accelerometer and gyroscope data. With a machine learning approach, the system is able to detect taps with a bagged decision tree classifier to almost 100%. Furthermore they were even able to guess individual letters in about 50% of the cases.

In this thesis, we are following the same basic idea, but for the scope of this bachelor's thesis, a training with machine learning algorithms to detect taps is unnecessarily complex. We can detect individual taps via a simple peak detection algorithm, as described by Palishkar et al. [28]. An example for peak recognition on the time series of sensor measurements is shown in Figure 2.2. In this Figure, ten peaks corresponding to a ten tap sequence are detected using Palischkar's algorithm. Eventual inaccuracies in identification of individual taps are not necessarily considered negative to the overall pattern recognition scheme, because those peaks are also part of the user's individual behaviour.

Palishkar's peak detection algorithm identifies peaks (also called spikes) in a given time-series of values. These peaks represent keystrokes or simply "values of interest" in our acceleration data. Between the detected peaks, no disturbance in the phones acceleration is found, i.e. the user did not touch or move the phone. A point in our data is a *local* peak, if it is a maximum value within a defined window and not too many other

points in the window have similar values.

Palishkar's algorithm is a parametric algorithm, that can be adapted to the individual structure of the time series data. The algorithm takes two additional parameters to the time series: the window size k around the peak to detect and a stringency h that rejects "low" peaks based on Chebyshev's inequality. Chebyshev's inequality states that for a random variable X with mean μ and standard deviation σ: $P[|X - \mu| \geq h\sigma] < \frac{1}{h^2}$. Based on this inequality, we can make a sophisticated guess, that a peak x with $|x - \mu| < (h * \sigma)$ is "small" in a global context and thus not a significant enough peak.

The window size k restricts the amount of peaks detected within k data points. To optimize the peak detection, we need to adapt k to the typical time of a key press. Is k too small, we might face the problem of detecting back-swings in the sensor data as additional peaks; is k too big, subsequent keystrokes might not be recognized. Therefore we analyzed typical usages to find a good k for our implementation in Section 3.2.4.

2.1.3 Features of keystrokes

Historically, the features of key strokes were only extracted from the events keyboards reported to the operating system. In example the X.Org Server, the de-facto standard input handling system in UNIX-like operating systems, handles a single key press via two separate events: A `KeyPress` event, whenever a key is pressed down and a `KeyRelease` event, when the key is lifted up again.

These two events can be measured according to several metrics, as defined by Dholi and Chaudhari [12] (cf. Figure 2.1). Furthermore, when considering more than two keystrokes, we can gather additional possible measurements:

- Overall typing speed, usually measured in Characters per minute (CPM)
- Overall typing rhythm and flow, measured in base frequencies
- Intensity of taps, measured by the amplitude of acceleration

To recognize typing patterns in arbitrary text, the typing patterns are usually broken down to di-graph, tri-graph or generally n-graph segments. This means, that the overall typing pattern is reduced to sequences of $2, 3, ..., n$ key presses and only the features of these n-graphs are analyzed.

In our approach, we are monitoring the user's input in a controlled environment, i.e. we only monitor input of the same sequence of characters, for example a password or a defined navigation sequence. This allows our approach to compare the whole sequence and we don't need to identify n–graphs in user input. Extending our implementation to also extract these graphs might be a next step to improve the generality of the implementation in subsequent work.

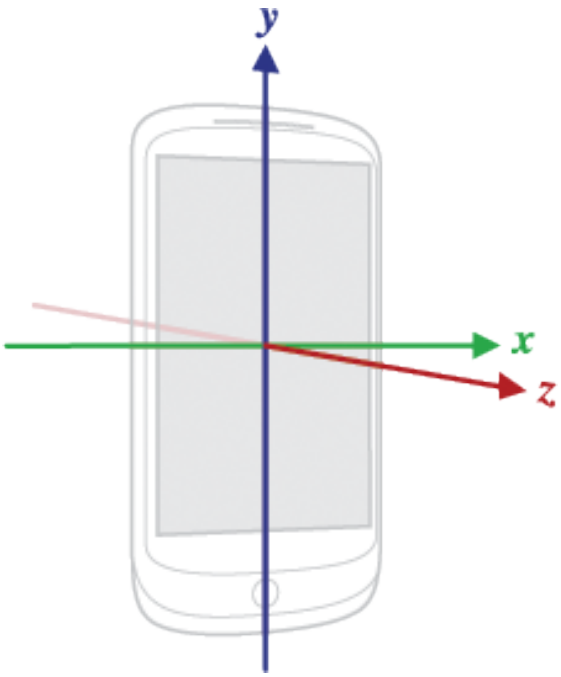


Figure 2.3: Inertial coordinate system of Android devices [5]

2.1.4 Adaption to phones

For the implementation of keystroke recognition on smartphones, we first need to identify how to efficiently identify individual keystrokes. In this approach, we are considering text input directly on a smartphone with no additional devices, i.e. the on-screen keyboard. In Android, the inertial coordinate system for the acceleration sensors is oriented as displayed in Figure 2.3. The coordinate system, according to which the acceleration sensors report their measurements is defined relative to the default orientation of the device and static, despite orientation changes of the devices display. The X-axis points horizontally to the right, the Y-axis vertically up and the Z-axis points towards the outside of the front face of the screen [5].

The main force of taps on a touchscreen is opposite to the direction of the Z-Axis as displayed in Figure 2.4. Thus, we can safely neglect the X- and Y-axis for our use-case of identifying individual taps on the screen.

Our basic approach in keystroke classification on smartphones is to use Palishkar's peak detection algorithm, as described in Section 2.1.2. We can apply this algorithm to the Z-axis acceleration sensor records of the phone, which promises sufficiently good data for individual taps.

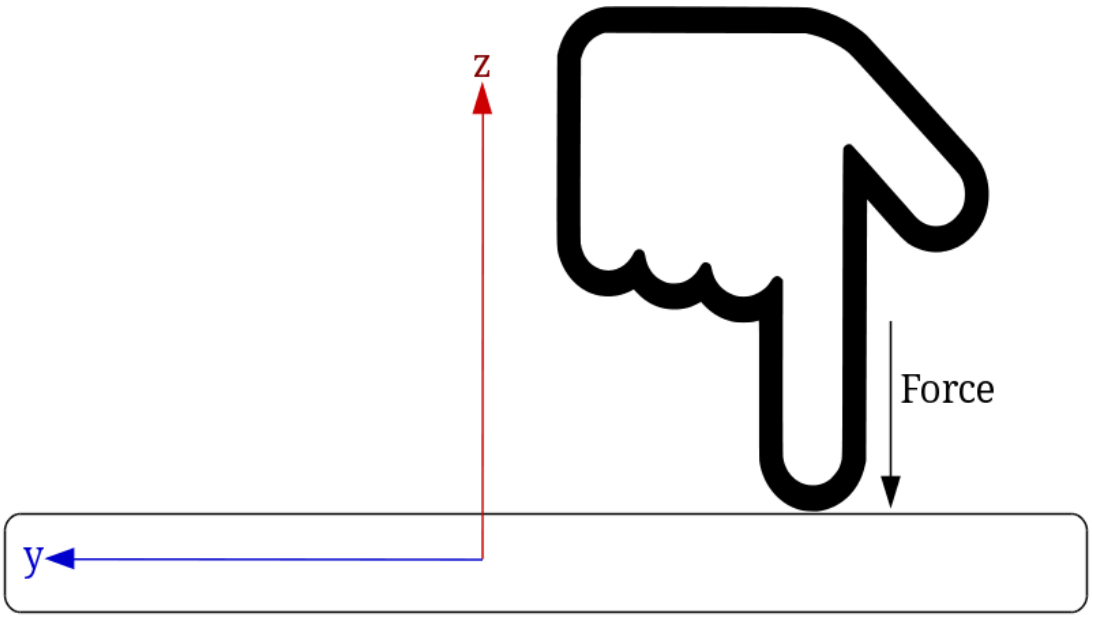


Figure 2.4: Model of touches on Android devices

2.1.5 Adaption to watches and keyboards

For recognition of key presses with smartwatches, we examined a scenario of a user wearing a smartwatch while typing on a physical keyboard. In this scenario, the X,Y-plane of the watch's inertial coordinate system is almost parallel to the keyboard (cf. Figure 2.5).

Wang et al. [36] discussed in their paper on MotionLeaks for MobiCom'15, that the key press timings on keyboards can be extracted by the Z-axis movement of the watch. When the user presses a key on the keyboard, the user's finger dips and the wrist also undergoes a partial dipping motion. This motion can be detected by the Z-axis acceleration, in combination with a peak detection algorithm, similar to the one used in Section 2.1.4.

Wang et al. also improved their peak detection algorithm by chaining a peak detection tool with a bagged decision classifier. Since their goal was to guess individual typed keys instead of analysing the pattern as whole, the additional computational overhead might be worthwhile. However, this does not hold true for our approach of matching the whole input pattern, thus we use a simple peak detection algorithm.

Additionally to keystroke pattern recognition the overall movement of the watch while typing can be used as an authentication vector (i.e. feature). Different users might move their hands differently for text input. However we are not aware of scientific studies

Figure 2.5: The coordinate system of a smartwatch while typing on a keyboard

measuring the effectiveness of this approach as of early 2016.

2.2 Gait recognition

Gait is defined as “the way in which a person (...) walks” in the Cambridge Dictionary. Various scientific papers analyzed the specifics of human gait [20, 23, 21] and concluded, that individual gait can be used as a biometrical recognition mechanism. The steps a user walks throughout the day can for example be used to generate user profiles and extract an identifying pattern.

As early as 1975, Johansson [20] has shown, that observers could identify individuals just by watching videos of lights mounted to joints of otherwise invisible walking people. Additionally, the observers were able to not only identify previously known people, but also identify the gender of unknown persons. However human gait is influenced by many more personal aspects, as the individual weight, leg length, posture and speed of walking. Thus gait patterns are highly individual and are usually unique.

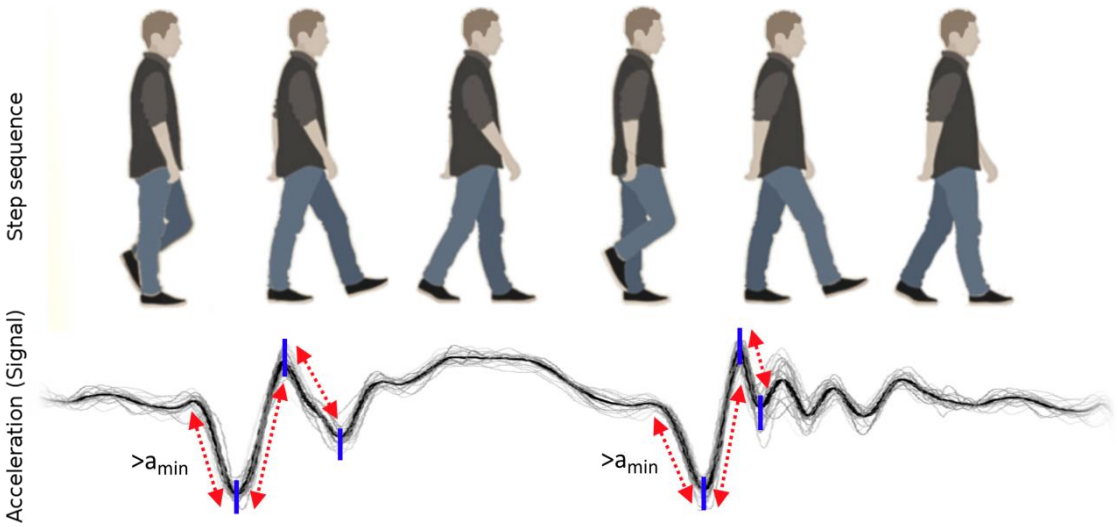


Figure 2.6: Motion circle and corresponding measured acceleration data [32]

2.2.1 Related work

Early attempts to gait recognition used video footage and moving light displays to extract the gait information. This approach worked quite well, but is largely unpractical nowadays, since face and shape recognition algorithms work even more precise on videos. Starting in 2005 with Mäntyjärvi et al. [25], researchers used accelerometers to extract gait information of users. These sensors were attached to different body parts, such as hip, arms and feet to evaluate the recordable data. Optimal positioning of these accelerometers is still disputed, but recent studies showed, that portable devices such as commercial phones [11] or smartwatches [21] are sufficiently good sensors for gait recognition. For gait recognition with mobile phones, Schmidtbartel [32] already implemented a framework to recognise specific user-device combinations. His model accumulated sensor data of the user's gait and aggregates a median step pattern, as shown in Figure 2.6. Each individual recorded step circle may vary due to signal noise or different user behaviour. Shown in the bottom half of the figure as individual lines, individual step circles are recorded in light grey. In black, the calculated median is shown. To calculate this value, multiple steps are clustered, according to the similarity of those steps. As a result, there might be multiple clusters of gait signals, i.e. for walking, jogging or running.

Schmidtbartel is using an Manhattan-Distance metric in his implementation, however other researchers [11] suggest, that Dynamic time warping (DTW) and an extension of DTW, called Cross DTW Metric result in even better gait recognition performance.

2.2.2 General limitations

To monitor the gait patterns of users, constant monitoring of the devices sensors is necessary, even though the user is not actively using the device. This prevents the device to go in so called “deep sleep” state where less energy is consumed. Since battery is a big concern on mobile devices, this is a major deal-breaker.

For authentication purposes, a timely responses to whether or not an authentication attempt was successful is required. However, gait is not available all the time and certainly not on demand. Prompting the user to take a walk to get access to his data is not a viable option.

2.2.3 Conclusion

As consequence of these limitations, we decided against using an gait recognition approach as personal authentication factor. Nonetheless, gait recognition might be an additional approach for intrusion detection, e.g. the device itself can recognise it being stolen by detecting other gait patterns. This allows the device to take counter-measures, e.g. lock itself, alert the user and activate “Find My Device” functionality. In contrast, gait recognition is not suitable for concrete, immediate authentication needs, as it is the vision of this thesis and Schmidtbartel already analyzed other usages in his thesis.

3 Implementation

For this thesis, we implemented app prototypes to demonstrate the capability of pattern recognition as a personal authentication factor. We chose Android as the main smart device platform, since access to development tools and documentation is freely available. Android applications are developed using Java, which allows to use many existing libraries. Android code also is portable across different form factors of devices, such as phones, tablets and smartwatches running Android-Wear.

The development and testing of the Android application was conducted on the author's personal devices, a OnePlus One and a Nexus 10. To be able to develop an Android Wear app, the Chair of Operating Systems kindly provided a Sony Smartwatch 3.

For the implementation we also used SQL as a platform-independent data storage. To visualize and plot the sensor measurements and their correlation to keystrokes, we used Python with the excellent *matplotlib* (cf. Figure 1.4).

3.1 Platform identification: Device vs. Server

As one of the first steps, we analyzed the target platform the application should run on. For an authentication scenario, we can identify two distinct platforms: A client, in our case an Android device, and a server, which wants to authenticate a user. For our approach, we can use both platforms, to implement pattern recognition of acceleration data. Thus we can gather arguments pro and contra each platform.

We could process all the data locally on the device, since we already need an Android App to record the accelerometer data. When we do the data processing client-side, we can keep the server application as small as possible. Since typical Android devices have multi-core processors, they also should be computationally powerful enough to process the data. In our test scope, all sensor processing work was handled fast enough to not create any user noticeable wait times (cf. Section 4.2). Moreover, decentralized data processing on the individual devices also has significant advantages of reducing server load. This allows the authentication approach to easily scale to a big number of users.

Local processing of the acceleration data also reduces the size of the data that needs to be transmitted to the server. The raw data can easily reach several megabytes, which is especially troublesome for mobile devices. When the networking connectivity of

mobile devices is slow, e.g. in cellular networks, raw data transmission can take several seconds.

There are also approaches to privacy protecting biometric authentication by locally generating cryptographic bio-keys [7, 35, 31]. Since handling personal identification information requires special precautions to not loose the data, these bio-keys can be used to implement so called “zero-knowledge proof of knowledge” protocols. This is especially useful, since only the information needed to verify the proof of knowledge needs to be stored on the server. If an attacker acquires this information, he can only verify the identity of the user but is not able to extract personal data about the user, thus preserving the users privacy.

A client-side data processing can also be used to authenticate the user locally, e.g. when entering the phones unlock code. This can be used for intrusion detection or for locking stolen devices. However, server side authentication is easier to deploy, since changes in the authentication algorithms only need to be made in a single place. Especially changes in the feature extraction seps are hard to deploy. Authentication spoofing is also harder, since an attacker can analyze locally installed apps, but does not have access to the server application.

For this thesis, we implemented a client-side authentication mechanism. In our opinion, the privacy of a user is very important and personal data should not be transmitted over a network, when better alternatives exist.

3.2 General purpose Android acceleration pattern detection module

Since we are creating two different apps, a normal Android and an Android Wear variant, sharing code among those apps is a necessity. To do this, we created a general purpose acceleration pattern detection module for Android. This also allows rapid prototyping and testing of the different approaches outlined in Chapter 2.

The main goal of this module is to allow easy integration into existing apps to enhance the authentication security without the need for specially crafted implementations. With our module, apps can compose their own pattern recognition implementations in a modular way, based on well known algorithms, as discussed in Section 2.

3.2.1 Sensor recording in Android

In Android, accessing the device’s sensors is managed by the `SensorManager` class. The process of obtaining the `SensorManager`, acquiring the default acceleration sensor and registering a custom `SensorEventListener` is shown in Listing 3.1. The mechanic of

Listing 3.1: Obtaining the default acceleration sensor data in Android

```
SensorManager mgr = (SensorManager) context
    .getSystemService(Context.SENSOR_SERVICE);
Sensor s = (manager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER));
mgr.registerListener(myListener, s, SensorManager.SENSOR_DELAY_FASTEST);
```

receiving data is then defined in this `SensorEventListener`, which is periodically called by the Android system with new data.

The rate at which the `SensorEventListener` gets callbacks is defined via the third parameter of the `SensorManager.registerListener()` function. In our case, we are using the fastest rate possible, since we don't want to miss even the slightest features of the movement pattern. Miluzzo et al. [26] have shown in their paper about guessing letters from device movement, that their results drastically improve with higher sensor sampling rate. Hence, we chose to poll the sensor at the fastest rate possible namely `SensorManager.SENSOR_DELAY_FASTEST`. In our tests, this corresponded to a sampling rate of about 200 Hz on a OnePlus One.

3.2.2 Sensor measurement framework

Within the module, we provide a simple way to record sensor values into a predefined data structure, called `SensorData`. This class, as well as all other classes defined to measure and record the sensors are organized in the `measurement` package.

The `SensorData`, as shown in Listing 3.2, consists of a two-dimensional array of floats, called `data`. The first dimension of this defines the direction of the sensor measurement, i.e. X- Y- and Z-acceleration, while the second dimension defines the series of individual measurements. We also record a timestamp of the individual measurements in the `timestamps` array. This is necessary, since the time between measurements can vary, depending on the current load of the processor. For example, if we access `data[0][41]`, we get the 42nd measurement of the X-acceleration in the series. The corresponding timestamp can be accessed via `timestamps[41]`.

Since arrays with static size are not suitable for dynamically building up data, we also defined a corresponding `SensorDataBuilder`, that uses lists of dynamic size to append new measurements. We can do this by calling the `SensorDataBuilder.append()` instance method, that dynamically grows the list as needed. When we completed recording of measurements, we can create a `SensorData` object of these measurements with the `SensorDataBuilder.toSensorData()` method.

The reasoning behind converting the data from lists to arrays is, that arrays have

Listing 3.2: Class `SensorData` containing the raw sensor readings

```
public class SensorData {

    public final float[][] data;
    public final long [] timestamps;

    public SensorData(float[][] data, long[] timestamps) {
        this.data = data;
        this.timestamps = timestamps;
    }

    public int getDimension() {
        return data.length;
    }
}
```

significantly less overhead in accessing random data as well as memory consumption. Also the preprocessing and feature extraction algorithms usually operate on simple arrays. So instead of converting the data for each step, we only use dynamic lists for buildup and use simple arrays afterwards.

3.2.3 Preprocessing of SensorData

To create meaningful and comparable sensor measurements, we needed to implement a preprocessing step after recording the sensor data. Since the measured acceleration includes gravity, we need to factor it out, depending on how the user holds the device.

To negate the effect gravity has on the measured data, we assume, that the device is relatively static in overall acceleration. This holds true for most applications, such as the device is lying on a desk or a user is carrying it around. We neglect the fact, that we cannot simply factor out gravity when we are measuring in a changing acceleration environment, which should happen infrequenty.

To factor out gravity, we normalize the measured data to get a mean value of 0. We can do this by simply calculating the mean value and shift all sensor values by the negative mean, which results in a mean of 0. This results in an overall formula of: $x_{new} = x - \mu$ with μ as the mean.

To enhance the sensor recordings, we also implemented smoothing functions, that are able to dampen sensor noise. As for this thesis, we implemented a simple moving

average and exponential smoothing filters. The simple moving average algorithm works by averaging a certain number of measurements in a so called “window size”. For comparison, we also implemented a simple moving average filter, which factors in the last smoothed value with a factor α with $0 < \alpha < 1$.

Since the rate of sensor measurements can vary with the processor load, we also could interpolate the measurements to a continuous function or simply a fixed rate. A technique to implement this would for example be cubic spline interpolation. However our sensor measurements were fairly regular in all conducted tests and sufficient for a proof-of-concept, with a standard variance in measurements $< 0.01ms$. In real world applications with varying loads and background activity during sensor measurements, a more sophisticated approach might be needed.

Since there might be multiple preprocessing steps needed, we also implemented a simple `ComposingPreprocessor`, which can combine multiple preprocessing steps to a single one. We apply the given preprocessing steps sequentially as specified. Thus we can for example first normalize the data to a mean of 0 and then smooth it with one of the implemented smoothing functions.

3.2.4 Feature extraction from SensorData

After the data is in a comparable shape, we can proceed to extract certain features. This is done by measuring the features of the data and storing those features in `FeatureVectors`. Those `FeatureVectors` have the advantage of being more condensed and smaller in file size than the raw data.

In our approach of detecting keystrokes, we first utilize a peak detection algorithm to locate the individual keystrokes as outlined in Section 2.1.2. For this, we are using Palshikar’s peak detection algorithm, which was implemented in the Fiji image processing library [28, 34]. Since this library is open-source, we adapted it to our needs. For best results, we used a window size of 67 measurements to detect keystrokes. The window size is based on the average keystroke frequency, which we measured to about 3 Hz. We then can calculate an appropriate size of the window to detect, as shown in Equation 3.1.

$$\begin{aligned} \text{windowsize} * \text{sampling frequency} &= \text{keystroke frequency} \\ x * 200\,\text{Hz} &= 3\,\text{Hz} \\ x &= \frac{200\,\text{Hz}}{3\,\text{Hz}} \\ x &= 66\frac{2}{3} = 66,\overline{6} \approx \underline{\underline{67}} \end{aligned} \tag{3.1}$$

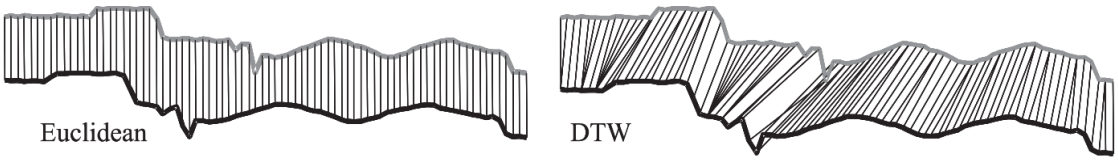


Figure 3.1: Comparison between a euclidean distance and a DTW distance [22]

Based on this algorithm, we implemented a `PhoneKeystrokeFeatureExtractor`, which classifies the features of detected keystrokes on conventional Android devices, such as phones or tablets. As a proof of concept, we are extracting the tap intensity and the individual interval of the keystrokes.

We did not implement a separate feature extraction mechanism for watches due to time limitations. Future approaches could be to record the lateral movement additionally to the tap intensity and the interval. This should result in better and more distinctive wear features.

3.2.5 Classification and machine learning

To allow an identification of users according to the entered data, we need a way to compare and classify the features mentioned above. As comparison algorithms, we can use an euclidean distance in n dimensions. In this distance, we compare two equal sized `FeatureVectors` by taking the square root of the distance of the individual features, squared. This formula is shown in Equation 3.2.

$$d(a,b) = \sqrt{(a_1 - b_1)^2 + \ldots + (a_n - b_n)^2} \tag{3.2}$$

Since our feature vectors are not necessarily equal in size, we decided to use a DTW distance measurement, as several other papers conclude, that DTW distances give significantly better results in comparing time series data [13]. This allows our implementation to be resilient to erroneously detected keystrokes. To visualize the idea, the difference in the two comparisons is displayed in Figure 3.1. In this Figure, two very similar time series are compared, but the euclidean distance is rather large, since the lower signal is shifted to the left and slightly stretched, where the DTW algorithm compensates the fluctuations.

The standard DTW algorithm uses dynamic programming, which fills a matrix of distance measurements and optimizes the path through it. There are also other implementations like SparseDTW or FastDTW available that require less computation time for big inputs. We decided to stay with a simple approach, since we already condensed the feature vectors to relatively small size and the dynamic programming implementation is fast enough for these small inputs.

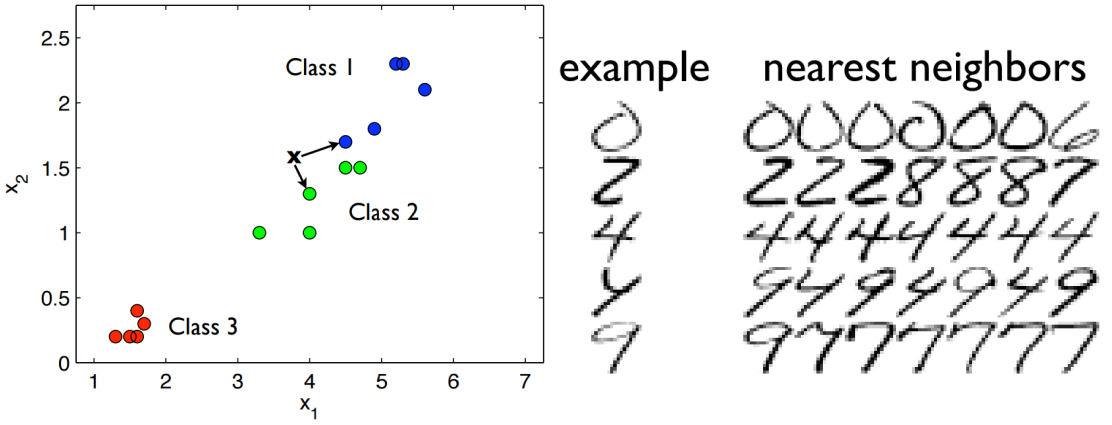


Figure 3.2: 2D visualization of the k-NN algorithm. On the left: Distances between points, defining nearness to neighbors; On the right: Nearest neighbors of hand drawn digits [24]

To classify the feature vectors, we implemented a k-NN algorithm according to Dudani's paper on this behalf [14]. With this algorithm, we can compare a new, uncategorized feature vector to all other vectors and record the distance to those vectors. We can then decide depending on the nearest neighbors (i.e. smallest distance) to which category (i.e. user) the new feature vector belongs. To do this, we look at the k vectors with the smallest distance to the new vector and count, which category is present most often.

To illustrate the k-NN algorithm, a two dimensional example, as shown in Figure 3.2, is better suited. On the left, we can see a plot of the already present data, represented as circles. These data-points are already categorized in three classes, shown as different colored points. To categorize a new data-point **x**, we can now calculate the distance of **x** to all other points via a distance measurement. The simple euclidean distance for two points is plotted as arrows in this example.

With the distances between the data, we can now determine the *nearest neighbors*, i.e. the data-points with the smallest distance. We can then decide, based on the majority of points in the k-nearest neighbors, to which class the new data-point belongs. The effect of this parameter k is visualized on the right side of Figure 3.2. In most examples, the nearest neighbors have a strong tendency towards a class, however, in edge cases k

can influence the final decision towards a certain class. Tests determining the optimal size of k can be found in Section 4.3.

Another approach would be to define and train neuronal networks to classify the available data. One open source framework for neuronal networks is Encog [30]. Due to the complexity and time requirement, this approach exceeds the scope of this Bachelor's thesis, but presumably is a promising future approach, which needs to be evaluated.

3.3 Data storage and processing

Since we are generating many individual data sets with recording acceleration data, we need a way to store the persistently. This storage is needed for later analysis, as the lifetime of Android apps is mostly controlled by the user. Therefore, we need a way to safely and persistently store the recorded data. With this data, we also can compare different preprocessing and classification approaches on the same test data.

3.3.1 SQLite Database

To store the data on the device, we use a SQLite database. SQLite is the standard Android SQL database for persistent storage on the device. SQLite also is an extremely lightweight database with little to no unnecessary features. Since we do not need extra database features except inserting and querying the data, SQLite is a good fit for our use case.

SQLite also stores the database in a single standardized file with a user-defined name, in our case `SensorMeasurements.db`. This allows to share the data between devices for debugging and data visualization. In our example prototype, the database is regularly copied to a folder accessible via USB, since the usual storage of the database is inaccessible for other processes running on the device. We then can access the database copy from a connected PC, where we can visualize the algorithms with a Python script.

The schema of the database is displayed in Figure 3.3. With this schema, we are storing the raw recorded data in so called "data sets". N of the data points in a data set form a measurement, which is linked to a single user. We also record the keystrokes in each of the measurements, when the user is using the on-screen keyboard. We then can correlate the "sensortime" we measured the individual acceleration in the data set with the "keystroketime" to examine, if we correctly detected a keystroke. We also deduced the average keystroke and sensor-measurement frequency used in calculation the parameters for the algorithms mentioned in the last sections from these recordings.

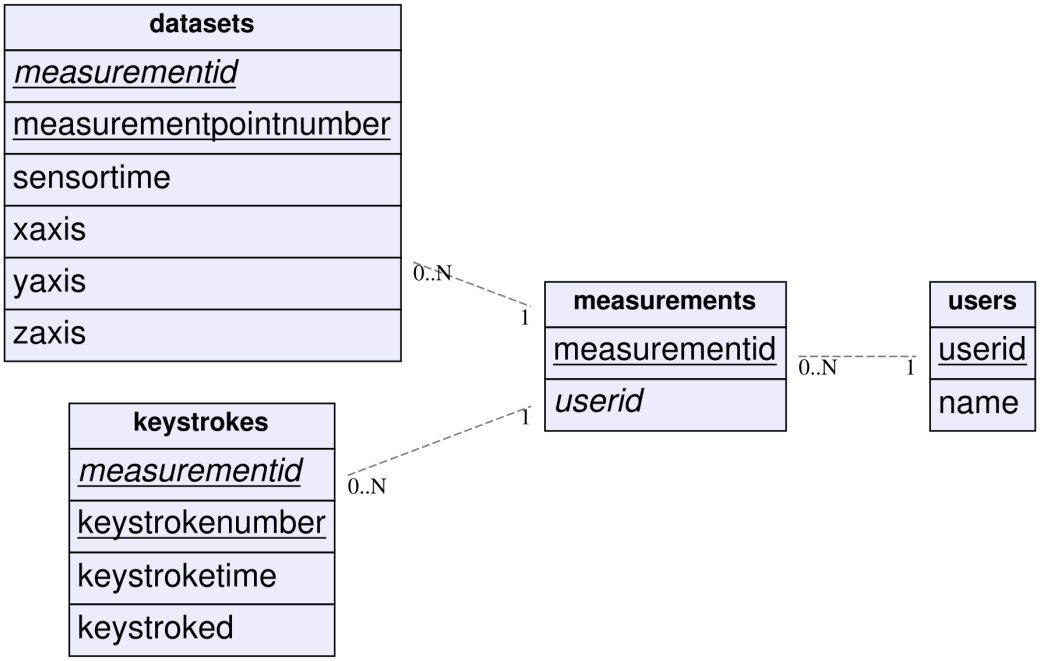


Figure 3.3: Schema of the SQLite database used by the prototypes

3.3.2 Background verification of patterns

Since the measurement data is persistently stored, we can also implement a background verification and identification of the users. These background identity checks would provide the benefits of additional authentication, without bothering or interrupting the user. Currently, this is only planned for future improvements. As of now, we directly check the measured data. However this can easily be implemented, depending on the individual use-case.

3.4 App prototypes

The individual app prototypes and source code are available online via GitHub (`https://github.com/pfent/GesturesID`) or via the attached CD. The structure of the files is as follows: In the “Android” folder, there is an Android Studio project called “GesturesID”, which contains the source code of the prototypes. In the same folder, there are also two `.apk` files, which can be installed on an Android, respectively an Android Wear device. This project can be built via the Android Studio Integrated Development Environment (IDE) or via executing the Gradle wrapper script `gradlew`.

The source code itself is structured in 3 different modules: `app`: the Android specific

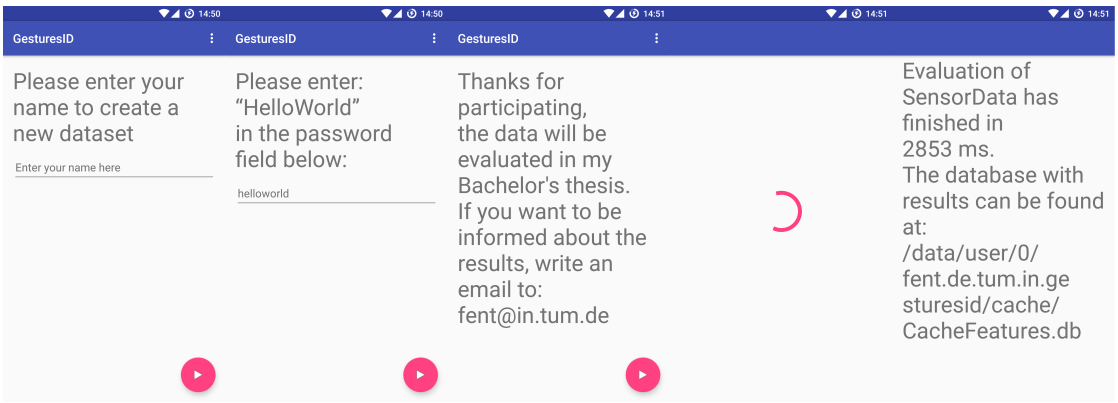


Figure 3.4: Screenshots of the prototype Android app, in sequence of execution. The left three screenshots show the interface to enter training data, the right two screenshots, how the data is being processed

source code; `wear`: the Android wear specific code and `sensorprocessing`: the code that can be used in both apps. Loosely speaking, all the portable classes are located in the `sensorprocessing` module. This includes the implementation for recording the sensors, preprocessing, analyzing and classification. The corresponding classes are structured in separate packages, which are accordingly named.

Additionally, the portable module also provides a `MeasurementManager` for persistently storing the data to a database. To interface with platform specific code, we also provide a Listener-interface, that is used for callbacks whenever a pattern has been recorded.

3.4.1 Android application

The standard Android app provides two Activitys. Both Activitys are launchable via Android's integrated app launcher, i.e. there are two different app icons. The first Activity "GesturesID" is used to record training data. This activity can be seen in the screenshots on the left of Figure 3.4.

In the current implementation, the user is first prompted to input a name, which is later used to identify the individual measurements. Afterwards, the user is prompted to enter a predefined sequence to generate acceleration data to test the pattern detection algorithm. The start and end of measurements is determined by a `PatternFocusChangeListener`. This listener analyzes the "focus" of the user and generates events, whenever the user taps into the `EditText` component to input text. When the user taps the enter key or proceeds to the next input field, this "focus" is lost and the listener fires an end event and we stop recording data.

In the second Activity, the data is processed with a specific configuration of al-

Listing 3.3: Minimum working example to extract keystroke features

```
SensorData data = new ComposingPreprocessor(
    new Selector(2), // Select Z-Axis
    new Normalizer()
).preprocess(
    MeasurementManager.getInstance(context).getSensorData(measurementID)
);
FeatureVectors vectors = new
    PhoneKeystrokeFeatureExtractor().extractFeatures(data);
```

gorithms described in the last sections. For the prototype, we are only using the normalized z-axis data. From this normalized data we then extract the features in a `PhoneKeystrokeFeatureExtractor`, which first determines the peaks (i.e. keystrokes) in this data. From these peaks we then can calculate the intensity of the individual taps and the intervals, in which the keys were stroked. A small code sample, how this could be implemented using our implementation can be seen in Listing 3.3.

Even though we implemented and tested smoothing of the sensor data, our tests showed, that smoothing the measurements did not provide better keystroke recognition rates, but decrease individuality of the tap intensity. With further investigation, we found that smoothing the measurements is unnecessary overhead, since the peak detection algorithm is specially engineered to ignore measurement noise, via factoring in the standard deviation.

After this learning phase, we then can classify new measurements, which need to be processed exactly the same as the training data. The classification process is displayed in Listing 3.4. For this prototype, we are using a k-NN classifier with a DTW distance measurement. The app then displays the time used to process all of the previously recorded measurements and displays the location, where it stored the intermediate results, as shown on the right of Figure 3.4.

3.4.2 Android Wear application

The Android Wear app is roughly built in the same shape as the Android app. The app has two launchable Activitys, one to record patterns and one to process them. However, the implementation is less elaborated, due to time constraints. The recording of patterns is manually initiated by clicking a button, as displayed in the left half of Figure 3.5. After starting the measurement, the button turns red and the measurement can be stopped by clicking the button again.

Listing 3.4: Minimum working example to classify given `FeatureVectors` according to previously learened `categories`

```
int classify (FeatureVectors[][] categories, FeatureVectors
    featureVectors)
    Classifier classifier = new kNNClassifier(categories, new
        dTWDistancer(3), 7);
    int category = classifier.classify(featureVectors);
    // category now contains the index of the category in categories[][],
    // where featureVectors belongs to
    return category
}
```

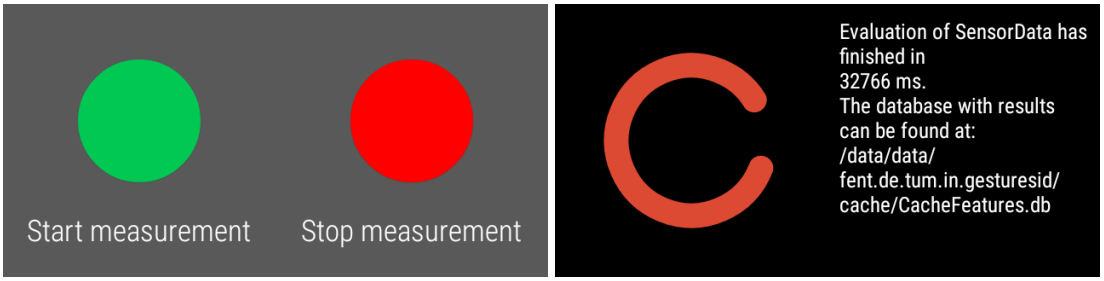


Figure 3.5: Screenshots of the prototype Android Wear app, in sequence of execution. The left two screenshots show the interface to record training data, the right two screenshots, how the data is being processed

While the Android app can automatically associate a user with a measurement by prompting to enter a name, this is not possible on Android Wear due to the lack of text entering methods on this platform. This would be possible, e.g. with a companion app on a PC or a phone, but has not been implemented yet.

The previously recorded measurements can be processed directly on the device, via the second Activity: `EvaluationActivity`, displayed on the right half of Figure 3.5. Currently the implementation only processes the measurement in the same way as the Android app. This might not result in the best user-detection results, but nevertheless can serve as a approximate performance measurement.

3.4.3 Limitations

In the current state, both prototypes work as a proof of concept. They allow to record data and compare different approaches and implementations of the algorithms described in Chapter 2. To authenticate and validate user identities in individual use-cases, additional work needs to be done, but can largely be based on the results of this thesis.

4 Evaluation

In this chapter, we evaluate the currently implemented prototypes in regards to efficiently, accurately, and practically extract behavioral patterns that identify individual users. First of all, we are measuring the runtime and performance of our implementation to verify, that the used approaches are practical. With a practical solution, we subsequently compare different parameters for the used algorithms to measure how accurately users can be identified with the app prototypes.

4.1 Test setup

For testing and comparing different approaches, we implemented a test setup that easily allows replication and validation of tests. In order to provide such a test setup, we are using test data generated by 5 students of the Android Practical course. The raw motion data of the users has been recorded to a database, which can be found on the attached CD, called `SensorMeasurements_tablet_usertest.db`, respectively `SensorMeasurements_wear_usertest.db`. In this database we recorded 5 users with 10 measurements each.

The Android measurements have been conducted on a Nexus 10 Android tablet. In this test, each user first entered his name, then ten times the sequence of characters: "helloworld". The recorded data then was saved to the `SensorMeasurements_tablet_usertest.db` database. For the Android Wear app, we used a Sony Smartwatch 3 to record the data. In our test setup, we did not individualize the measurements promtly, but simply edited the `userID` in the database according to the users afterwards. In the Android Wear test, the users each entered the character sequence: "correcthorsebatterystaple" ten times.

To replicate these tests, the databases can be copied to the apps databases folder as `SensorMeasurements.db`. Executing the `EvaluationActivity` will process the data and write any intermediate results to a `CacheFeatures.db` database, located in the cache folder of the app. The Android app also logs detected users to Android's logging system, which can be viewed with `adb logcat`.

The code test setup is to first run the preprocess and feature extraction steps on all measurements. With these features we can train our classification algorithm. For this

we use all but one data set for each user. With this trained classifier, we can measure our detection rate for the remaining data set.

4.2 Runtime efficiency

Since the current setup is mainly used for debugging, we are logging the results and itermediates to an additional database. This results in a relatively poor performance of 4.3 measurements per second on a OnePlus One smartphone. However this test is not limited by the processor, but by the database and the storage write speeds. In an optimized environment, where we turn of writing to the database and thus keep all data in memory only, we get much better results. Our tests showed, that with optimizations, we can reach $\sim$21 measurements per second.

We now reached an almost 100% CPU utilization of one core. A possible additional optimization would be parallelization. An evaluation showed, that the preprocessing and feature extraction can completely be done independently from one another. This could potentially speed up the learning phase by a factor of 4 x.

On Smartwatches, significantly less processing power is available, thus processing the measurements should take proportionally longer. Expectedly, measurements on a Sony Smartwatch 3 showed a throughput of 1.8 measurements per second with logging enabled. This is roughly proportional to the clock speed of the smartwatch's processor. Potential optimizations are the same as on a smartphone, i.e. disable database writing and parallelization, with potential speedups of $\sim$5 x.

As a note, transferring the data from the watch to the smartphone for processing would not speed up the computation time, since measured Bluetooth transfer rates of $\sim$15 MBps are slower than the processing throughput on the watch itself.

4.3 Optimal parameters for accuracy

In our test setup, there are in total four tunable parameters, that influence the classification results. Starting with feature extraction, we can set the window size and the stringency for the peak detection algorithm. The window size is essentially the time between two detected peak and thus limits the maximum detected features. On the other hand the stringency defines, how "big" a peak needs to be in order to be recognized.

For classification, we can influence the window size of the DTW algorithm, i.e. how many false detections we can tolerate. We can also set the k parameter in the k-NN algorithm, influencing cluster sizes.

We found, that the window size of 67 we calculated in Equation 3.1, is in fact outperforming bigger or smaller window sizes. With the stringency however, it was not exactly clear, what the optimal value looks like. Palshikar [28] recommends, that the stringency h should typically be $1 \leq h \leq 3$. Comparing the overall results of the stringency, we found that a value of 2 resulted in the best extracted features. The DTW window size did not affect the results positively with bigger window sizes, so we used a relatively small value of 3. With the nearest neighbor algorithm, we got the best results with a $k = 7$.

With these optimal parameters, we reached an peak detection rate of 80%, i.e. 4 out of 5 users correctly classified.

5 Conclusion

As summarization, our implementation serves as a proof of concept. We reached the goal of identifying users based solely on acceleration data and showed how to efficiently extract individual features from acceleration data, that was gathered on mobile devices.

5.1 Current state

The tests we conducted had a relatively small sample size, thus the results are not universally applicable. Nevertheless, a 80% detection rate of our system shows, that it can be used for safer and more convenient authentication mechanisms.

The performance of processing and classification of acceleration data is in a realistic range for practical use. Entering a password takes several seconds, i.e. the same duration needed to process and train a classification algorithm with >100 previous measurements. For real usages, the processing and training phase does not need to take place every time, but can be cached and afterwards simply read from storage.

Even without caching and therefore calculating everything each time, our system performs reasonable well on smartphones as well as on a smartwatch. Authentication on the Sony Smartwatch 3 takes aproximately 30 seconds, which is not great, but still in an acceptable time-frame for watch apps. Our implementation is however not suitable for smartrings, because with even lower powered processors, they are probably too slow for data processing directly on the ring. Since smartrings are still in a concept phase, there are good chances that future development in low power processor speeds will result in smartrings with comparable processing speed to smartwatches nowadays.

5.2 Future prospects

For future projects, our implementation can be used with small adaptions to the individual use case. Nevertheless, there are still many possible extensions and improvements left for future work. Currently all of the parameters are statically determined and might not perform the same on all device configurations, especially with fluctuating sensor recording rates. An attempt to work with different recording rates would be interpolation of measurements, e.g. cubic spline interpolation to get continuous sensor values.

Overall the Feature extraction steps are in a good shape, but especially implementing specific feature extractions for Android Wear should make it more practical. The biggest potential improvement left to evaluate are neuronal networks to classify measurements. Neuronal networks resulted in immense improvements for speech and image recognition in the last years and many neuronal network implementations have been open sourced recently.

Glossary

Acceleration sensor An acceleration sensor (also called accelerometer) is a device that is capable of measuring the forces accelerating devices. These forces are often called “g-forces” from the gravitational force..

Activity An Android Activity is the basic user interface of an Android app. Activities can usually be launched from other apps or the start menu.

app An application program. A computer program designed for a specific type of application.

Dynamic Time Warping An algorithm measuring the similarity between time series data as a distance, compensating variations in time or speed.

euclidean distance A metric measuring the distance between data points, based on the square root of the sum of the squared absolute differences.

Fragment A Fragment in an Android context is a dynamic component of an Android Activity. Fragments are usually used to build virtual layouts that are larger than the physically displayed layouts and provide standardized interfaces to dynamically load content that probably is being showed next..

Keystroke dynamics Individual characteristics in typing, e.g. key strokes, that can be used to identify users.

Manhattan-Distance A metric measuring the distance between data points, according to the sum of the absolute differences between coordinates.

package A Java package is a structure to organizing Java code into name-spaces. Classes in a package are usually dependent on each other, but preferably not on other classes..

Pattern recognition Pattern recognition is a machine learning technique that can classify patterns based on previously learned patterns.

Pattern recognition Pattern recognition is a machine learning technique that can classify patterns based on previously learned patterns.

Acronyms

API Application Programming Interface.

APK Android app package file.

CPM Characters per minute.

DTW Dynamic time warping.

GSM Global System for Mobile Communications, originally Groupe Spécial Mobile.

IDE Integrated Development Environment.

k-NN k-Nearest Neighbors.

MFA Multi-factor authentication.

OTP One Time Password.

SaaS Software as a service.

SQL Structured Query Language.

TUM Technische Universität München.

UMTS Universal Mobile Telecommunications System.

USB Universal Serial Bus.

WiFi A local area wireless networking technology.

List of Figures

Listings

Bibliography

[1] https://oneplus.net/one. Accessed: 2016-02-05.

[2] http://www.sonymobile.com/de/products/smartwear/smartwatch-3-swr50/. Accessed: 2016-02-05.

[3] http://smartyring.com/. Accessed: 2016-02-05.

[4] F. Aloul, S. Zahidi, and W. El-Hajj. "Two factor authentication using mobile phones." In: *Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference on*. IEEE. 2009, pp. 641–644. URL: http://staff.aub.edu.lb/~we07/Publications/Two%20Factor%20Authentication%20Using%20Mobile%20Phones.pdf.

[5] Android Open Source Project. *Android's SensorEvent API reference*. https://developer.android.com/reference/android/hardware/SensorEvent.html. Accessed: 2016-01-05.

[6] D. Bartmann. "PSYLOCK—Identifikation eines Tastaturbenutzers durch Analyse des Tippverhaltens." In: *Informatik'97 Informatik als Innovationsmotor*. Springer, 1997, pp. 327–334.

[7] A. Bhargav-Spantzel, A. Squicciarini, and E. Bertino. "Privacy preserving multi-factor authentication with biometrics." In: *Proceedings of the second ACM workshop on Digital identity management*. ACM. 2006, pp. 63–72.

[8] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[9] S. Block and A. Popescu. *DeviceOrientation Event Specification*. http://www.w3.org/TR/orientation-event/#devicemotion. Accessed: 2016-01-04.

[10] N. L. Clarke and S. Furnell. "Authenticating mobile phone users using keystroke analysis." In: *International Journal of Information Security* 6.1 (2007), pp. 1–14.

[11] M. Derawi and P. Bours. "Gait and activity recognition using commercial phones." In: *computers & security* 39 (2013), pp. 137–144.

[12] P. R. Dholi and K. Chaudhari. "Typing pattern recognition using keystroke dynamics." In: *Mobile Communication and Power Engineering*. Springer, 2013, pp. 275–280.

[13] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. “Querying and mining of time series data: experimental comparison of representations and distance measures.” In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1542–1552.

[14] S. A. Dudani. “The distance-weighted k-nearest-neighbor rule.” In: *Systems, Man and Cybernetics, IEEE Transactions on* 4 (1976), pp. 325–327.

[15] evelyn and starbug. “Basteltips Biometrieversand.” In: *die datenschleuder* 92 (2008), 56f.

[16] T. Fiebig, J. Krissler, and R. Hänsch. “Security impact of high resolution smartphone cameras.” In: USENIX Association, 2014.

[17] R. S. Gaines, W. Lisowski, S. J. Press, and N. Shapiro. *Authentication by keystroke timing: Some preliminary results*. Tech. rep. DTIC Document, 1980.

[18] Gartner. *Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 3rd quarter 2015*. `http://www.statista.com/statistics/266136/`. Statista, Accessed: 2016-01-16.

[19] IDC. *Market share of smart wristwear shipments worldwide by operating system in 2015 and 2019*. `http://www.statista.com/statistics/466563/`. Statista, Accessed: 2016-01-09.

[20] G. Johansson. “Visual motion perception.” In: *Scientific American* (1975).

[21] A. H. Johnston and G. M. Weiss. *Smartwatch-Based Biometric Gait Recognition*. `http://storm.cis.fordham.edu/gweiss/papers/btas-2015.pdf`. Accessed: 2015-12-10.

[22] E. Keogh and C. A. Ratanamahatana. “Exact indexing of dynamic time warping.” In: *Knowledge and information systems* 7.3 (2005), pp. 358–386.

[23] L. Lee and W. E. L. Grimson. “Gait analysis for recognition and classification.” In: *Automatic Face and Gesture Recognition, 2002. Proceedings. Fifth IEEE International Conference on*. IEEE. 2002, pp. 148–155.

[24] M. S. Lewick and C. Mellon. *Artificial Intelligence: Representation and Problem Solving - Clustering*. `https://www.cs.cmu.edu/afs/cs/academic/class/15381-s07/www/slides/042607clustering.pdf`. Accessed: 2016-01-20. 2007.

[25] J. Mäntyjärvi, M. Lindholm, E. Vildjiounaite, S.-M. Mäkelä, and H. Ailisto. “Identifying users of portable devices from gait pattern with accelerometers.” In: *Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on*. Vol. 2. IEEE. 2005, pp. ii–973.

[26] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. “Tapprints: your finger taps have fingerprints.” In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM. 2012, pp. 323–336.

[27] F. Monrose and A. D. Rubin. “Keystroke dynamics as a biometric for authentication.” In: *Future Generation computer systems* 16.4 (2000), pp. 351–359.

[28] G. Palshikar et al. “Simple algorithms for peak detection in time-series.” In: *Proc. 1st Int. Conf. Advanced Data Analysis, Business Analytics and Intelligence*. 2009.

[29] Parks Associates. *Nearly 60% of U.S. broadband households use OTT video services, but “Account Sharing” is prevalent*. `http://www.parksassociates.com/blog/article/cus-2015-pr11`. Accessed: 2015-12-10.

[30] H. Research. *Encog Machine Learning Framework*. `http://www.heatonresearch.com/encog/`. Accessed: 2016-01-15.

[31] A. Ross and A. Othman. “Visual cryptography for biometric privacy.” In: *IEEE transactions on information forensics and security* 6.1 (2011), pp. 70–81.

[32] N. Schmidbartl. “Analyse zur Identifizierung von Benutzern/Geräten anhand verschiedener Faktoren und Integration in ein Framework.” Accessed: 2016-01-06. MA thesis.

[33] The Manhattan district attroney’s office. *Report on Smartphone Encryption and Public Safety*. `http://manhattanda.org/sites/default/files/11.18.15%20Report%20on%20Smartphone%20Encryption%20and%20Public%20Safety.pdf`. Accessed: 2016-02-05. 2015.

[34] J.-Y. Tinevez. *PeakDetector.java*. `https://code.google.com/p/fiji-bi/source/browse/src-plugins/FlowMate_/fiji/plugin/flowmate/analysis/PeakDetector.java?r=0ec4620b8e4aaebd183c2b57c89390595f574564`. Google Code, Accessed: 2016-01-14.

[35] E. Verbitskiy, P. Tuyls, D. Denteneer, and J. Linnartz. “Reliable biometric authentication with privacy protection.” In: *Proceedings of the 24th Benelux Symposium on Information theory*. 2003.

[36] H. Wang, T. T.-T. Lai, and R. Roy Choudhury. “MoLe: Motion Leaks through Smartwatch Sensors.” In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM. 2015, pp. 155–166.