Adam Roberts

Pico for iOS

Computer Science Tripos – Part II

St John's College

May 19, 2017

Proforma

Name:	**Adam Roberts**
College:	**St John's College**
Project Title:	**Pico for iOS**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**11967**
Project Originator:	Dr F. Stajano
Supervisor:	Mr S. Aebischer and Dr D. Llewellyn-Jones

Original Aims of the Project

To create a Pico client for iOS devices that could be used with the existing Pico system.

Work Completed

I created a working implementation of the Pico client on iOS. My implementation has a run time that is around 3.75 times faster than the current implementation for Android devices. I also added TouchID as a new security feature, removing an attack vector on the system. I also implemented an elliptic curve cryptography library in Swift that, to my knowledge, is the first of its kind written purely in Swift.

Special Difficulties

None.

Declaration

I, Adam Roberts of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed Adam Roberts

Date May 19, 2017

Contents

List of Figures

Acknowledgements

I would like to thank Seb Aebischer and David Llewellyn-Jones for all of their help during the course of my project.

Chapter 1

Introduction

My project concerns building a client for the Pico system, allowing users to remove passwords from their lives. I have succesfully created an iOS app that is compatible with the existing Pico system and runs much faster than the existing Android app. My app meets all of my success criteria and I implemented one of my proposed extensions; TouchID support.

1.1 Motivation

Passwords are the most widely used authentication system in the world. Commonly a user will authenticate with some service by providing a username that can be publicly known and a password that must remain private. This is a very simple system but it is inherently insecure. Passwords are shared secrets between the user and the service, and a security failure at either end can lead to the system being broken. Attackers have two main vectors to break a password-based authentication system.

The first is technical: by exploiting vulnerabilities in the system an attack may be able to access secure information as the service will have to store some kind of password data. A well secured service would store hashed, salted and peppered passwords but this is not always the case.

The second is human nature. Since passwords are, in general, chosen by humans, an attacker can exploit the predictability of human nature to guess the user's password. This could be anything from using personal information specific to that user to guess the password, to using the password stolen from another attack knowing that many users will reuse the same username and password combination for many services.

The most common technique used by attackers to gain a user's passwords is phishing, wherein an attacker will craft a webpage to look the same as the official page in the hope that a user will enter their credentials and submit their own password to the attacker. Phishing alone costs the U.S. economy an estimated $2bn per year [1], and in recent months there have been high-profile data breaches of political parties in the US and

French elections, where in both cases the attackers are believed to have gained access to email accounts using phishing techniques [2].

Even though passwords have several serious issues, they continue to be widely used due to their convenience. Any security system has a major trade-off between security and convenience. This should seem intuitive; an email account with no password would be extremely convenient but insecure, and an email account that could not be accessed is extremely secure but not particularly convenient. An ideal system would therefore be one that gives users the best possible combination of security and convenience.

Attempts to improve password systems have been largely carried out by individual institutions to increase their level of security, but in general this has decreased the convenience of a password scheme to a point where users will actively try to break the systems. A common example in industry is where employees are forced to change their passwords on a monthly basis. This leads to users creating a pattern of passwords, for example, in month one using "password1", in month two using "password2", etc. This actually gives a lower level of security than a password system without these rules as it gives the illusion of safety that can make users use less caution when protecting their passwords [3]. It is clear that there is an opportunity for a password replacement system to solve the aforementioned problems.

1.2 Overview of Pico

Pico [4] is a project that aims to replace passwords with a system that is both highly secure and convenient. It relies on the principle of a user carrying a physical authentication token that is used to prove the identity of the carrier. This token connects to the service that the user is authenticating to, creates a secure channel and then uses cryptography to prove the user's identity.

The eventual aim of Pico is to completely replace passwords in the day-to-day usage of most people. Pico provides an easy-to-use system for a user and a simple-to-implement interface for developers. It eliminates the need for passwords completely. While password wallet apps may appear similar at first glance as they also don't require the user to remember any secrets, they still use passwords behind the scenes that leaves many of the security flaws associated with passwords.

The current client side implementation of Pico is an Android application. While Android maintains a dominant market share, in the UK Android and Apple's iOS combine to make up 99.1% of the smartphone market, so having an implementation on each would cover the vast majority of users. This would be helpful as the goal of the Pico project is to become ubiquitous and replace password schemes wherever possible, and the "chicken-and-egg" problem is a major hurdle that needs to be overcome for this to become a reality: nobody will use the app if no services allow Pico logins, and no service will allow Pico logins if nobody uses the app.

1.3 Related Work

There have been many attempts at password replacement schemes in the past using wildly different methods of authentication. These have ranged from fingerprint scanners to encrypted password managers to sheets of transparent plastic with patterns printed on them that can be placed on a monitor screen and traced with the cursor [5].

The vastly different methods put forward by different people show how difficult it has been to find a replacement to passwords that can be more secure but at least as convenient. Certain schemes have been successful in specific areas, for example in recent years fingerprint scanners have become common on smartphones, but there has yet to be a successful password replacement scheme that works across all levels of security.

The Pico team in the computer laboratory has already completed a large amount of work on the Pico project. A working prototype client exists as an Android app, and there are implementations of the service-side applications, including a browser plug-in that enables Pico on standard websites. This meant I was able to use parts of the Pico libraries in my implementation.

Chapter 2

Preparation

2.1 Structure of Pico

Pico is composed of two parts: a client and a service. In most cases the client is a physical device owned by a user and the service will be a computer or website the user wishes to authenticate to.

There are two stages required to use Pico to authenticate to a service. The first is pairing, in which the service and the client exchange public keys that are then stored and used as identifiers. Later, when the client wishes to authenticate to the service the authentication process is invoked, where a secure channel is established and each of the service and the client prove their identities.

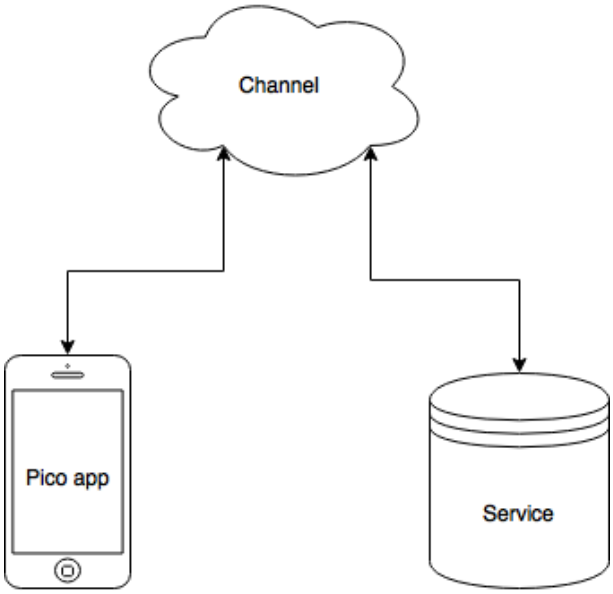

Figure 2.1: A visualisation of a channel.

The service and client authenticate over a channel as shown in Figure 2.1. A channel is just an abstraction for data communication, in the current implementation either as

HTTP/HTTPS requests over the Internet or as Bluetooth messages. My implementation uses HTTP requests as they are platform-independent and Bluetooth channels were not fully supported by the Pico platform whilst I was creating my implementation.

When using an HTTP channel, a QR code is used for the first stage of communication. This QR code is created by the service and displayed on screen and contains the information needed to start either the pairing or authentication procedure, depending on which the user wishes to invoke. This information will include the service's identity public key (or a hash of that key in the case of authentication), a URL that can be connected to as the channel, an extra data field that can be used to convey any data that specific service may need to communicate to the client and vice-versa. For example, to use Pico to log into a computer, during the pairing procedure the computer will send the client an encrypted version of the password needed to log into the computer. During authentication this password will then be sent back and used to authenticate the user to the computer.

2.2 Requirements Analysis

Since I was working with an already developed system, client implementation would have to follow a certain structure. To help with this, I prepared a plan for how the app should be structured to work with the Pico back-end. My app would consist of four main parts:

- **Code scanner** - This would scan the QR codes presented by the service, deserialise them and pass the required information onto the next stage.
- **Database** – The database would contain all of the information created by each pairing, store it and allow it to be easily retrieved when authentication occured.
- **Sigma prover** – The sigma prover is the part of the app that executes the protocol used to authenticate to a service. This is the core of the app, and would have to complete the SIGMA-I protocol, which uses Eliptic Curve Diffie-Hellman (ECDH) and the Eliptic Curve Digital Signature Algorithm (ECDSA). ECDH is used to generate a shared secret between the client and the service and ECDSA is used to create a signature that proves ownership of a key.
- **iOS wrapping** – All of the parts would have to be wrapped in iOS specific code, allowing them to be executed on an iPhone.

There will be two main workflows in my app, one for pairing with a service and one for authenticating with that service.

The main difference between the two workflows is whether to store the data generated by the protocol, or to use data already created during the protocol. Depending on the type of the QR code scanned, one of two possible paths are taken. This is shown in Figure 2.2

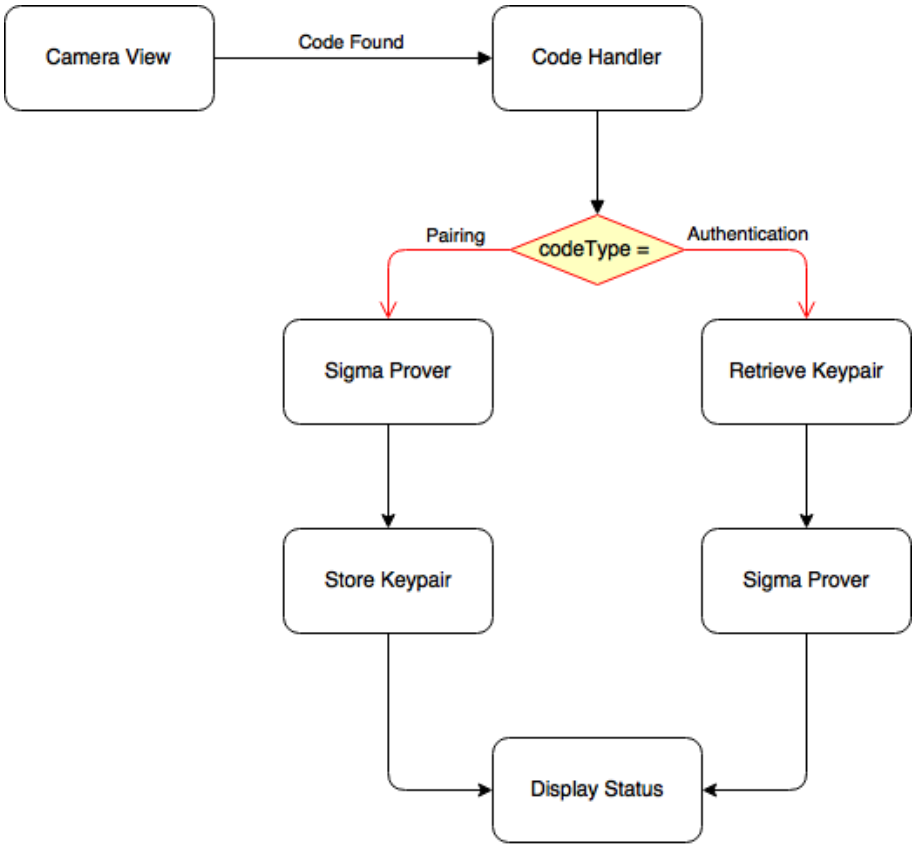

Figure 2.2: High-level view of a Pico workflow.

- Key-pairing case – the app generates a new identity key pair and initiates the SIGMA-I Protocol with the service. If the protocol completes successfully, the new pairing can be stored in the database. Once this is completed, a box pops up on the phone's screen giving the user the status of the pairing.
- Key-authentication case – the app checks whether a pairing exists for the service. If a pairing exists then it is loaded, if not then the user is notified. Now the SIGMA-I protocol can be initiated. The user is again notified of the status of the authentication.

2.3 Developing for iOS

iOS is the operating system that runs on Apple's mobile devices. For my app, I targeted the iPhone platform with version iOS9. This means the app can run on any iPhone or iPad with iOS9 or higher installed natively. The only SDK able to deploy to iOS is XCode which limits app development to Swift and Objective-C.

Developing on iOS revolves around Views. A View is a visual representation of some underlying data. Each View is controlled by a `ViewController` that acts as a middleman, updating the view when the data changes and updating the data when the user interacts with the View.

The way iOS apps are structured makes the Model-View-Controller (MVC) design pattern seem like a natural choice to structure apps. Here, the Model refers to the underlying data, while the View and Controller are obvious. Even though my app would not have multiple Views, I structured my app with the MVC pattern as it gives a very intuitive development workflow.

2.4 Language choice

As mentioned earlier, there are two languages that are supported for development on iOS, Swift and Objective-C. For the majority of the application-specific development I chose to use Swift 3.0. This is due to the fact that it is Apple's preferred language choice for development and it is also easier to use and more readable than Objective-C. For the pure Pico logic, I worked in C. This was possible because Objective-C is a superset of C, so C code can be run by wrapping the functions with either iOS specific Swift or Objective-C code. There are three reasons for choosing C, firstly, the SigmaProver method is the main method call used when both pairing and authenticating with a service. This is the only time-sensitive portion of my application so it is important to be as efficient as possible. Since C is a more low-level language than either Swift or Objective-C, performance tends to be better. Secondly, writing the core in C makes it much easier to interact with the required libraries, namely the libPico library and OpenSSL. Finally, writing the Pico logic in C means that my code is able to be added to the libPico library and then run on many platforms. If I wrote the core in Swift it would only be able to run on Apple systems, whereas a C implementation can run on any platform with a C compiler, including any future custom built Pico hardware.

I originally intended to use the Java Pico library, JPico, instead of libPico, as JPico has much more functionality than libPico. To attempt this, I used Google's J2ObjC tool that converts Java source code to Objective-C source code. While this seemed like a good idea at first, it became clear early during development that it would not work. J2ObjC does not support cryptographic primitives, as the automated translation can not guarantee that the generated code will be secure. To use these generated sources I would have to implement a lot of the cryptographic functions myself in Obective-C. Instead of this, I chose to just write the cryptographic functions in C, so they would be much more portable.

2.5 Libraries used

During the course of development, it became clear that a few libraries would be required for my app. Since the vast majority of core libraries on iOS were built in Objective-C, their Swift interfaces are often over-complex for their usage. To solve this I used the database wrapper FMDB to work with the built-in SQLite on iOS, and I used SwiftyJSON which is a Swift wrapper for the built-in Objective-C JSON library.

Neither of these libraries are required *per se*, but using them allowed me to focus on the core parts of my project rather than getting caught up in needlessly complicated syntax.

The libPico library contained the necessary code for a Pico service. I expanded the library to give it the ability to function as a dual client-service library, during which I used some of the data types and methods to avoid code duplication. The libPico library has a dependency on OpenSSL, which was also required for my application.

OpenSSL contains the cryptographic primitives required for the SIGMA-I protocol which can be considered the heart of Pico. Although I did implement ECC in Swift, I could not find a way to get this to work with the existing Pico system, so OpenSSL was used in my final version of the app.

I also used SMP which is an arbitrary precision integer library in Swift, as this was required for my implementation of ECC.

2.6 Software development methodology

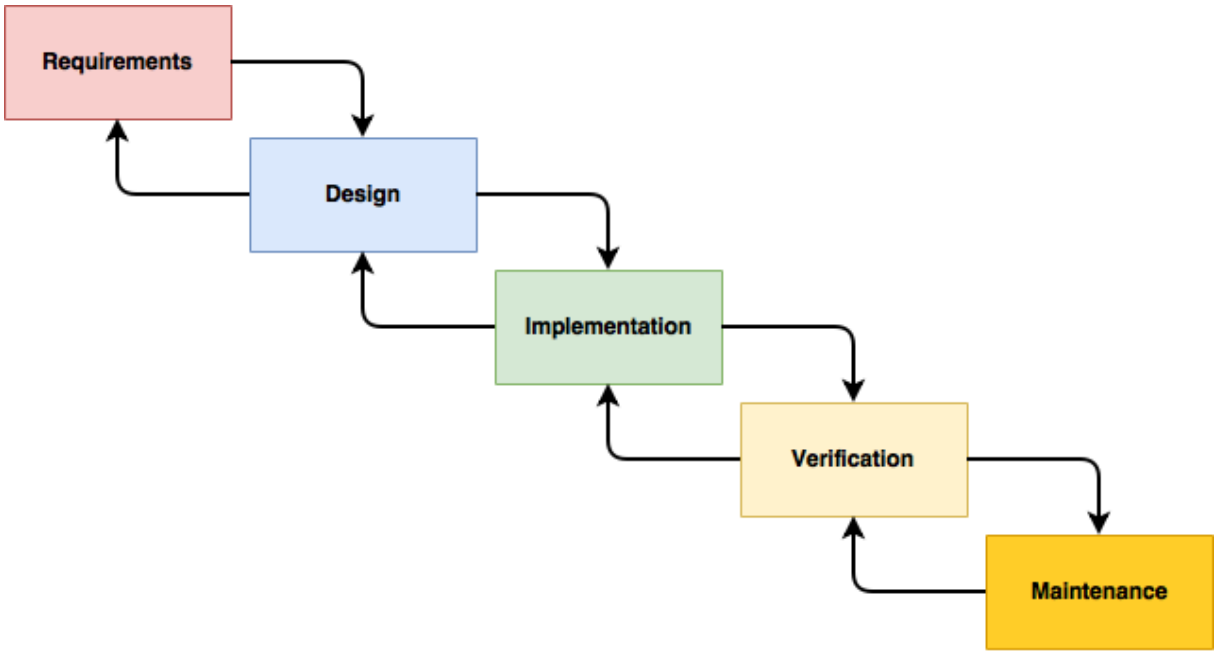

Figure 2.3: The Sashimi Waterfall model.

The software development methodology I followed is the Sashimi Waterfall (or Overlapping Waterfall). As shown in Figure 2.3, the Sashimi Waterfall differs from standard waterfall by allowing me to move back a level at any point. This worked well since the project structure was fairly clear from the start of the project as my app would have to be compatible with an existing system, but even though I had a good idea of the structure of my app from the start, some minor design changes would be made during development due to the fact that a lot of the finer details of Pico lacked documentation so I would have to work out how to do things as I got to them.

My requirements analysis completed the *Requirements* and *Design* steps of the Sashimi Waterfall. My implementation completed *Implementation* and my evaluation completed *Verification*. No *Maintenance* is needed as my app has not been deployed yet.

2.7 Revision control and backup strategy

I used GitHub [6] as my primary version control strategy. XCode provides very good support for GitHub, so this reduced the whole git process to commit, push and pull buttons. I would push my work to the online repository after I added any feature or decided to stop working for the day. Since GitHub is out of my control, and therefore a point of failure, I made sure to keep back-ups of my work on two flash drives, which were updated on a weekly basis.

Chapter 3

Implementation

3.1 QR code scanning

3.1.1 QR codes in Pico

QR codes in Pico are a form of optical channel that begin the first stage of either the pairing or authentication processes. These codes are representations of a JSON dictionary that contain the data needed to perform the relevant action based on that code. The action needed to be taken by the scanning device is determined by the code's type, which is given by the `'t'` value of the JSON dictionary. An example of one of these codes can be seen in Figure 3.1

Figure 3.1: A key-pairing QR code.

My implementation focused on two types of QR code that are fundamental to Pico, key-pairing and key-authentication codes. The key-pairing QR code is scanned when a

user first uses a service. This lets the service and client create a record of each other's identity that can then be used to complete the future authentication procedure. The key-authentication QR code allows the user to authenticate to a service that they have created a pairing to. This is analogous to creating an account on a service with a key-pairing, and logging in using that account with a key-authentication.

To see the data that each QR encodes, it will be useful to see examples of a JSON dictionary from each type of QR code. The data contained in a key-pairing code can be seen in Listing 3.1 and the data from a key-authentication code can be seen in Listing 3.2.

```
{
    "sig":"MDUCGAIk6BvthUwbEevs+rs+7q4N8OtqW/
          JANAIZAKrxoCmE9Excqz5QC7Kas7RA48nkfq18AA==",
    "td":{},
    "spk":"MEkwEwYHKoZIzj0CAQYIKoZIzj0DAQEDMgAEZZ01uagRK8iut
          fkn8zJgjc5JNVSQr9rNnkdTaS5vRI/7oBCm7H8kSd9qley8NO+q",
    "ed":"",
    "sn":"PicoServerTest",
    "sa":"http://rendezvous.mypico.org/channel/
          1aceb84542aecc2cf97d27fb3644d70b",
    "t" :"KP"
}
```

Listing 3.1: Example JSON dictionary represented by a key-pairing code.

`sig` and `td` are not relevant to my implementation.

`spk` represents the long-term public key of the service. This is used as an identifier for the service as the service can use its long term private key to sign messages to prove its identity.

`ed` represents extra data. This is essentially a slot for any data that may be needed specifically for that service.

`sn` is the name of the service, which is used by the client to identify the service in a human-readable form.

`sa` is the address of the channel to be used to connect between the service and the client.

`t` is the type of the QR code that has been scanned.

```
{
    "td":{},
    "sc":"iTPmcJmRSCJfEkwiL0PP27JcELjoW6q73gRrLnUFD6A=",
    "ed":"",
    "sa":"http://rendezvous.mypico.org/channel/
          8abcc34dae67f53cc6d987b967dc8480",
    "t" :"KA"
}
```

Listing 3.2: Example JSON dictionary represented by a key-authentication code.

You can see here that for an key-authentication QR code, most of the values are also present in the pairing code. The only new field is the `sc` field, which stands for service commitment. This contains a SHA-256 hash of the service's identity public key. This is so a client can only find out the service's public by pairing. During the pairing process the client can calculate the service's commitment and store this along with any data from the pairing.

3.1.2 AVFoundation code scanner

To scan the QR codes in my app, I decided to use iOS's built-in `AVFoundation` framework [7]. This allows for quick and reliable code scanning in iOS without using any external libraries. To achieve this, I overrode the `viewDidLoad` method of my main view controller as shown in Listing 3.3. This is called whenever the view controller loads, and since my app only has this view controller, this code will be run whenever the app opens.

```
1  override func viewDidLoad() {
2      ...
3      captureSession = AVCaptureSession()
4      videoInput = AVCaptureDeviceInput(device: videoCaptureDevice)
5      captureSession.addInput(videoInput)
6      ...
7  }
```

Listing 3.3: View controller code to open a connection to the phone's camera.

This creates a `captureSession` object that is a representation of a video stream currently being captured by the camera. I could then use this object to display the camera's view on the screen. This creates a simple UI which was my aim for the project as this allows a simple 'point and shoot' use case. Now that I have a video feed from the camera, I can use the built in functions to scan the feed for a valid QR code. This is accomplished by creating a connection from the video feed to the main Dispatch Queue. Dispatch Queues in iOS are a way of executing asynchronous operations without writing multi-threaded code. This is done using a producer-consumer relationship, where in my code the video feed is the producer, and the code handler is the consumer. This is shown in Listing 3.4

```
1      captureSession.addOutput(metadataOutput)
2      metadataOutput.setMetadataObjectsDelegate(self, queue:
3          DispatchQueue.main)
4      metadataOutput.metadataObjectTypes = [AVMetadataObjectTypeQRCode
      ]
```

Listing 3.4: Connecting the camera to the main Dispatch Queue.

When a QR code is found, the code handler method will be called with the data from the QR code as an argument.

3.1.3 Pico code handler

Having received the result from the Dispatch Queue, I have a textual representation of the data encoded by the QR code. The first thing to do is to convert this to the JSON type and read the type of the code, because as shown earlier each type encodes different data so the type must be known before accessing the correct elements of the JSON dictionary. Once the type has been read, I could take a code path for each of the possible types.

Since my application only uses key-pairing and key-authentication codes, there are two branches to take here, one for each code. The branches here will complete any actions required outside of the `SigmaProver`, which is the cryptographic heart of my application. By having this code outside of the `SigmaProver`, the `SigmaProver` can be used for any code type as long as the relevant extra code is added to the code handler.

A channel is opened before any code type-specific action is executed. As discussed earlier, a channel is an abstraction for data communication within the Pico system. Since all of the QR codes in Pico start communication between the client and a service, this step will have to be taken any time a valid code is scanned. The libPico library provides a method of connecting to a channel, so the address from the QR code is passed to the channel connection function.

Key-pairing

If the code that has been scanned is a key-pairing code the first action taken by my app is to generate a new long-term identity key pair. There are two reasons for this, the first is that creating a new identity key-pair allows the app to generate multiple pairings with the same service, allowing one user to have many accounts.

The second, more important reason is that creating a new key-pair for each pairing minimises the damage caused by any breach in security. While theoretically it is not (currently) computationally possible to calculate an elliptic curve private key based on a public key there are other possible exploits an attacker could use to find a user's private key. If the iOS platform is susceptible to attacks an attacker could use this to gain access to the phone's flash storage and therefore the database of keys stored on the device. In this scenario, if the app uses the same private key for all pairings, then the attacker would have access to all past and future pairings and that specific Pico client would become unusable. With a randomly generated key being used for each pairing the attacker would still be able to access past pairings but these could be changed once the attack was detected. Randomly generated keys also provide the benefit of malicious providers not being able to track users across different services. Since large companies control multiple popular services, they may find it useful to track users across all their products or alternatively law enforcement could request them to track users. Neither of these scenarios are particularly beneficial to the privacy of the user.

The key-pair is generated using the libPico library. I will explain this process in detail in the next section.

Once the key-pair is generated the commitment of the service's public identity key is calculated. This is because when the app scans a key-authentication code the public key commitment is provided, so by using that as the key in a database means the pairing information can quickly be retrieved.

The newly generated keypair, the generated commitment and data contained in the extraData field of the original code are then held together in a KeyPairing object. If the pairing process is successful, this object is stored in the database for later retrieval during authentication.

Key-authentication

If the code that has been scanned is a key-authentication code the only extra action that needs to be taken is retrieving the pairing information from the database. This gives the required information to complete the SIGMA-I protocol, including any service specific extra data given during the pairing process.

3.2 Elliptic curve cryptography

3.2.1 Elliptic curve basics

This section describes an elliptic curve library that I implemented in Swift. For the final version of my project, I used OpenSSL instead of this library for reasons discussed later. Regardless, understanding ECC is a very important part of my project.

Elliptic curve cryptography forms the basis of password replacement in Pico, before I continue it will be helpful to understand the mathematical basis of ECC.

A basic understanding of finite fields would be helpful, but all that is needed for the mathematics I present is the understanding that $\mathbb{F}_p$ is a finite field over some prime p with the operations addition, $+$, and multiplication, $\bullet$. All operations in the field are performed modulo p.

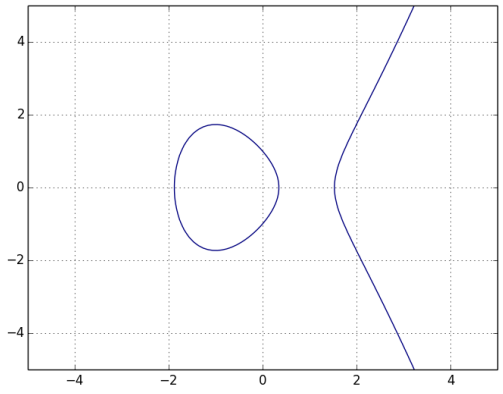

Figure 3.2: An elliptic curve with parameters $a = -3$ and $b = 1$.

My implementation was done using affine co-ordinates so the following will be based on that as opposed to projective co-ordinates that can also be used.

At this level, it is enough to describe an elliptic curve as a plane curve described by the equation

$$y^2 = x^3 + ax + b \tag{3.1}$$

where a and b are real numbers. This is a simple representation of an elliptic curve known as a Weierstrass Equation. An example can be seen in Figure 3.2.

The only constraint we have is that the curve must not be singular, again at this level there is no need to go deeply into the mathematics but it means that the discriminant of the curve must not be equal to 0.

The discriminant of the curve is given by

$$\Delta = -16(4a^3 + 27b^2) \tag{3.2}$$

which means that

$$4a^3 + 27b^2 \neq 0 \tag{3.3}$$

To use these curves on a computer we will limit them to a finite field over some prime p. This allows us to perform functions that are computationally easy one way, but hard to reverse. This also limits the curve to the set of points with integer coordinates, which are easy to work on with standard computer architecture.

Limited over a finite field, $\mathbb{F}_p$, the curve becomes a set of points. We must also add a point at infinity that I will represnt as $\mathcal{O}$, which acts as an identity value for addition. This set can now be written as

$$\{(x, y) \in (\mathbb{F}_p)^2 \mid y^2 \equiv x^3 + ax + b \pmod{p}, 4a^3 + 27b^2 \not\equiv 0 \pmod{p}\} \cup \{\mathcal{O}\} \tag{3.4}$$

While there are in infinite number of choices of a and b, the vast majority of implementations use curves specified by the American National Institute of Standards and Technology (NIST). This is because these curves have been specifically chosen to be both efficient and resistant to known attacks.

To use a curve the following parameters are required; the prime field p, a, b, a base point G, the order of the base point n and the cofactor h. The order of the base point is the smallest positive integer such that $n \bullet G = 0$ and the cofactor is the order of the curve divided by the order of the sub-group created by the base point. For efficiency reasons it is useful to have this as small as possible (usually one) as this allows us to create more keys from one base point.

To use this curve for cryptographic purposes, basic arithmetic must be performed on the points in this set, specifically point addition and point multiplication. First, it is required to understand the notion of division in a finite field. In a finite field we can divide f by g by multiplying f by the multiplicative inverse of g, that is $f \bullet g^{-1}$.

To find the multiplicative inverse of a number in a finite field modulo p, I implemented the extended Euclidean algorithm. The algorithm is shown in psudeocode in Algorithm 1.

Algorithm 1 Extended Euclidean Algorithm

```
1:  function inverseMod(k, p)        ▷ Find the multiplicative inverse of k modulo p
2:      t ← 0
3:      oldT ← 1
4:      r ← k
5:      oldR ← p
6:      while r ≠ 0 do
7:          q ← oldR/r
8:          (oldR, r) ← (r, oldR − q ∗ r)
9:          (oldT, t) ← (t, oldT − q ∗ t)
10:     if r > 1 then return There is no inverse for k in p
11:     if oldT < 0 then oldT ← oldT + p
12:     return oldT
```

3.2.2 Point addition

Now that division is implemented, I could move onto the elliptic curve specific operation of point addition. The addition of two points, A and B on an elliptic curve is defined as the negation of the point of intersection of the straight line defined by the points A and B. This is shown graphically in Figure 3.3.

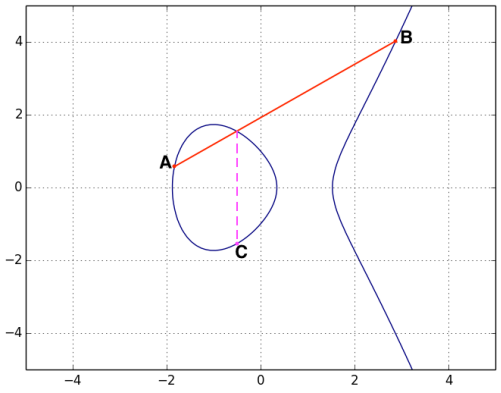

Figure 3.3: The addition of points A and B result in C, the negated intersection point.

As shown in Figure 3.4, negation of a point on an elliptic curve is done by simply negating the y component in the affine representation.

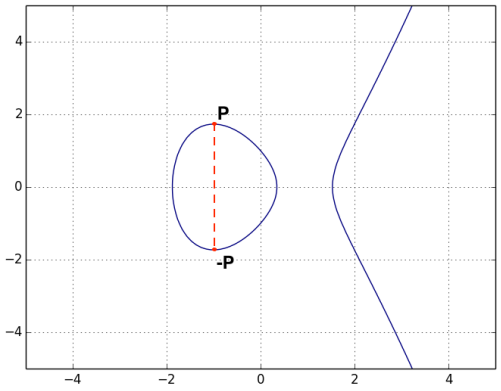

Figure 3.4: The negation of the point P results in the point $-P$, which is P mirrored across the x axis.

Since in my implementation points are represented as a basic class with x and y values, the function to compute this was equally basic and can be seen in Listing 3.5.

```
1 func pointNeg(point:Point) -> Point{
2     return Point(x: point.x, y:(-point.y))
3 }
```

Listing 3.5: Function to negate an elliptic curve point.

Due to the properties of elliptic curves, the line defined by any two distinct points A and B where $A \neq -B$ will only have one intersection point. This leaves three edge cases that must be dealt with.

The first is where $A = -B$. In this case the addition is $A + (-A)$ which since we are in a field we define this as $\mathcal{O}$.

The second case is where either of the arguments are $\mathcal{O}$. In the case, the result is the other argument, or $\mathcal{O}$ if both arguments are $\mathcal{O}$.

The final case is where the points are not distinct, which is known as point doubling. In this case, since there is no single line defined by the points, we define the line as the tangent to the curve at that point. This will leave a single point of intersection that can be mirrored to get the result of the addition. Given two points (x_1, y_1) and (x_2, y_2) the sum of the two can be calculated by

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned} \tag{3.5}$$

where m is the gradient of the line defined by the two points. In the case that the points are distinct, this is given by

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{3.6}$$

In the point doubling case, m is

$$m = \frac{3x_1^2 + a}{2y_1} \tag{3.7}$$

which is the first derivative of the Weierstrass form of the curve. Since my implementation is on a curve over $\mathbb{F}_p$ the results of the addition have to be taken modulo p and the division is done with the multiplicative inverse. My implementation of point addition is shown in Listing 3.6.

```
1  func pointAdd(point1: Point, point2: Point, curve: Curve) -> Point{
2      let pointAtInfinity = Point(x: 0,y: 0)
3
4      if(point1 == pointAtInfinity){ return point2}        //P + 0 = P
5      if (point2 == pointAtInfinity){return point1}
6
7      let x1 = point1.x, y1 = point1.y, x2 = point2.x, point2.y
8
9      if(x1 == x2 && y1 != y2){                            //P + -P = 0
10         return Point(x: 0,y: 0)
11     }
12     var m = BInt(0)
13
14     if (x1 == x2){
15         m = (3*x1*x1 + curve.a) * inverseMod(k: 2 * y1, p: curve.p)
16     } else {
17         m = (y1 - y2) * inverseMod(k: x1 - x2, p: curve.p)
18     }
19
20     let x3 = (m * m - x1 - x2) % curve.p
21     let y3 = (y1 + m * (x3 - x1)) % curve.p
22
23     return Point(x: x3, y: -y3)
24
25 }
```

Listing 3.6: Addition of two points on a curve over $\mathbb{F}_p$.

3.2.3 Point scalar multiplication

Scalar multiplication is the last operation needed now to be able to form cryptographic system. The naive implementation of repeated addition would work but its run time would be exponential in the length of the scalar. Instead, it is possible to perform scalar multiplication in linear time. To do this I used the double-and-add algorithm. As the name implies, this works by doubling the point in the same amount as the binary representation of the scalar. For example, to calculate 13 • P with a doubling function $d()$

$$13 \bullet P = (2^3 + 2^2 + 2^0) \bullet P = d(d(d(P))) + d(d(P)) + P \tag{3.8}$$

I have presented this algorithm in pseudocode in Algorithm 2, which my implementation performs modulo p.

Algorithm 2 Double-and-Add

```
1: function scalarMultiply(P, k)                ▷ Point P, scalar multiplier k
2:     add ← P
3:     out ← 0
4:     i ← 0
5:     for i to m do
6:         if k_i = 1 then                      ▷ k_i is the ith lowest endian bit of k
7:             out ← pointAdd(out, add)
8:         add ← pointAdd(add, add)
9:     return out
```

3.2.4 Key-pairs

Elliptic curve cryptography is a public key system. This means that each user has two keys, a public key PK and a secret key SK, the combination of which is known as a key-pair. The public key can be published and used by others as it is computationally infeasible to infer the secret key from the public key.

Any curve used as a base for an elliptic curve cryptography implementation requires a base point. This base point is a generator for $\mathbb{F}_p$. In this system, the secret key is a random integer and the public key is the base point multiplied by this integer. Creating a key-pair like this allows for some interesting operations to be performed that I will describe in the next subsection. My implementation can be seen in 3.7.

```
1  func generateKeyPair(curve: Curve) -> KPair{
2      var privKey = BInt(0)
3      while(privKey = 0){
4          let initKey = randomBInt()
5          let privKey = initKey % curve.n
6      }
7      let pubKey = scalarMultiply(k: privKey, point: curve.basePoint,
       curve: curve)
8      return KPair(privKey, pubKey)
9
10 }
```

Listing 3.7: Generation of a key-pair.

Now that I have implemented basic elliptic curve operations I can use these to perform the cryptographic operations required by Pico.

3.2.5 Elliptic curve Diffie-Hellman

Pico uses Ephemeral Elliptic Curve Diffie-Hellman (EECDH) to generate a shared key over an insecure channel. EECDH is similar to standard Diffie-Hellman except instead of integer exponentiation modulo some prime we can do point multiplication over $\mathbb{F}_p$. Due to the associativity of multiplication in a finite field, it is possible for two parties to exchange public keys to create a shared secret that can not be calculated by an observer.

Consider two parties, Alice and Bob, who wish to created a shared key so they can encrypt messages between each other with some cipher. An attacker, Mallory, is monitoring the channel they are communicating on and records all messages sent.

Both Alice and Bob generate a keypair, and share their public keys with each other. Alice receives Bob's public key, PK_B. This key was created by multiplying the base point of the curve by Bob's secret key, so it can be written as

$$Curve_{BasePoint} \bullet PK_B \tag{3.9}$$

Alice can now multiply this by her private key, PK_A. She now has the point given by

$$Curve_{BasePoint} \bullet PK_B \bullet PK_A \tag{3.10}$$

Bob can also do the same with his secret key and Alice's public key giving

$$Curve_{BasePoint} \bullet PK_A \bullet PK_B \tag{3.11}$$

Due to the associativity of multiplication we can see that

$$Curve_{BasePoint} \bullet PK_A \bullet PK_B = Curve_{BasePoint} \bullet PK_A \bullet PK_B \tag{3.12}$$

Therefore Alice and Bob have calculated the same point without exchanging their private keys. Mallory cannot use any of the information she has recorded to work out the shared secret due to the elliptic curve discrete logarithm problem that is given $k \bullet Q \pmod p$ find k. This is believed to be computationally hard, although it has not been proven. From this shared point Alice and Bob can simply take the value of the x component as a shared key that can be used to generate a key for a cipher to encrypt messages. My implementation of this can be seen in Listing 3.8.

```
1 func generateSharedSecret(selfKey: KPair, recievedPublicKey: Point,
    curve: Curve) -> BInt{
2     let sharedSecret = scalarMultiply(k: selfKey.privKey, point:
    recievedPublicKey, curve: curve)
3     return sharedSecret.x
4 }
```

Listing 3.8: Generation of a shared secret using ECDH.

Ephemeral-ECDH differs from ECDH in the fact that random keypairs are generated for each session of communication. This means that two parties communicating will generate different shared secrets each time they open a connection between each other. This provides forward security, because if Mallory works out the shared secret for one session, she cannot use this information to break the key for the next session.

3.2.6 Elliptic curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA), is a variant of DSA that is used to create a proof of ownership of a secret key without giving any information about the key itself. This is useful to me as secret keys are used as identifiers for actors in the Pico system. There are two steps that are needed, signature generation and signature verification. Both of these require a hash function, for which I used SHA256 as that is the hash function the Pico system uses. My implementation of signature generation can be seen in Listing 3.9 and my implementation of signature verification can be seen in Listing 3.10.

```
1  func signMessage(privateKey: BInt, message: String, curve: Curve) ->
       (BInt, BInt) {
2
3      let hash = hashMessage(message: message, curve: curve)
4      var r: BInt = 0
5      var s: BInt = 0
6      var k: BInt = 0
7
8      while(r != 0 || s != 0){
9          while(k == 0 || k >= curve.p){
10             k = randomBInt(bits: curve.bitLength)
11             k = k % curve.p
12         }
13         let point = scalarMultiply(k: k, point: curve.basePoint,
       curve: curve)
14
15         r = point.x
16         s = ((hash + r * privateKey) * inverseMod(k: k, p: curve.n))
       % curve.n
17
18     }
19
20     return (r, s)
21
22 }
```

Listing 3.9: Generation of a signature.

It is extremely important for the k value to be a cryptographically secure random number. If it is not there are attacks that can recover the private key used to generate the signature. This was most famously done to find the private key Sony used to verify PlayStation 3 games, which allowed attackers to run unauthorised software and pirate games [8].

```
1  func verifySignature(publicKey: Point, message: String, r: BInt, s:
       BInt, curve: Curve) -> Bool{
2      let hash = hashMessage(message: message, curve: curve)
3      let w = inverseMod(k: s, p: curve.n)
4      let u1 = (hash * w) % curve.n
5      let u2 = (r * w) % curve.n
6      let point1 = scalarMultiply(k: u1, point: curve.basePoint, curve
       : curve)
7      let point2 = scalarMultiply(k: u2, point: publicKey, curve:
       curve)
8      let checkPoint = pointAdd(point1: point1, point2: point2, curve:
        curve)
9
10     if( (r % curve.n) == (checkPoint.x % curve.n)){
11         return true
12     }
13     else {
14         return false
15     }
16
17 }
```

Listing 3.10: Verification of a signature.

3.3 SIGMA-I Protocol

3.3.1 Overview of the protocol

The SIGMA-I protocol [9] is the protocol used by the Pico system to communicate between a client and a service. It is a method of performing ECDH safely, even in the prescence of an adversary. A high-level view of the protocol can be seen in Figure 3.5.

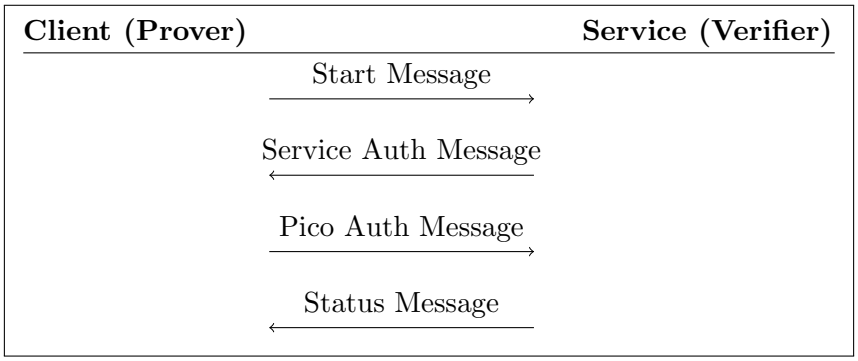

Figure 3.5: An overview on the SIGMA-I protocol.

The protocol has two actors, a verifier and a prover. The client app on the phone is the prover, as it is proving its identity and the service is the verifier, as this is checking the credentials and authenticating the client if they are valid. The service I used to test my implementation was the `picotest` application which uses the libPico library to generate QR codes on the command line. It can be executed in either pairing or authentication mode and will generate the relevent QR code. The application works by generating a QR code that contains a URL. When the client sends a `start message` to that URL, the application calls the `SigmaVerifier` from libPico which allows the SIGMA-I protocol to begin.

The premise of the SIGMA-I Protocol is to use ECDH to generate a shared secret. This secret is then used as a key to encrypt messages sent between the prover and the verifier. It then uses ECDSA to prove the identity of both participants. This works on an insecured channel because as discussed before, an adversary eavesdropping messages would not be able to generate the shared secret and would therefore not be able to read the messages sent between the prover and verifier.

For my implementation of the `SigmaProver` I used C and OpenSSL instead of my Swift library implemented in the previous section. There are a few reasons for this:

- **Compatibility** – The existing Pico system uses OpenSSL/BouncyCastle on the service end. Both of these use ASN1 encoding when serializing and de-seralizing keys sent over the network. This means to get my code to work with the current Pico system I would have to implement an ASN1 compiler/decompiler that would have taken a significant amount of time to implement and is not particularly relevant to my project.

- **Speed** – My implementation runs in Swift using a pre-made library that wasn't specifically designed with elliptic curve cryptography in mind, which leads to slower performance than OpenSSL (although after I completed my evaluation, it became clear that if I used my elliptic curve implementation, run time for the app would be comparable to the current Android implementation of Pico). Also, there are several patents relating to commonly used algorithms for fast elliptic curve arithmetic. While I am fairly confident that since software patents are un-enforceable in the EU this would be fine, litigation was out of scope for my project so I decided against it.
- **Portability** – The current libPico library is written in C using OpenSSL. By creating my implementation the same this allows my code to work freely with the rest of the system on any device that supports C. Since Swift is relatively unsupported on non-Apple systems, implementing the `SigmaProver` in Swift would only allow it to officially run on iOS or OSX.
- **Security** – Pico has to be secure otherwise it is useless, OpenSSL is a well tested, open source library that has a lot of trust in the security community.

SIGMA stands for **SIG**nature and **MA**c, as the protocol uses these to authenticate a user to a service. The I stands for **I**nitiator, as the protocol provides identity protection, that is an eavesdropper cannot gain knowledge of the identity of either participant, with stronger identity protection for the client as the client can choose whether or not to reveal its own identity after it has gained knowledge of the identity of the service.

3.3.2 Data Structure

For my implementation I took advantage of several existing data structures in libPico and extended them to suit my needs. The general overview is that there is a C struct, `Shared`, which is used as a container for all of the data needed by the protocol.

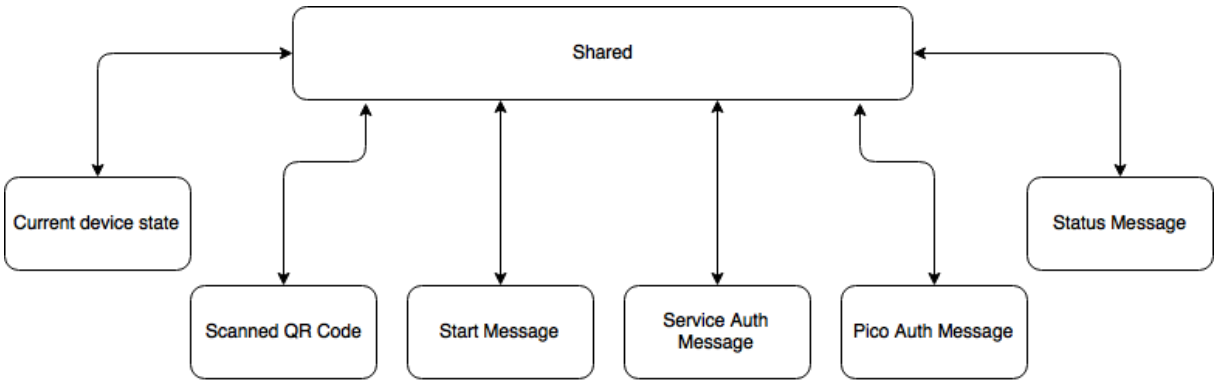

Figure 3.6: Visualisation of the data structure behind the `SigmaProver`.

As data is needed to create messages it is read from `Shared`, and as data is received from messages over the channel that data is written to it.

A single instance of this struct is created when the app is first loaded the data it contains is changed when needed, such as changing the Pico's identity keypair when a key-pairing QR code is scanned. `Shared` is then passed between functions as the protocol progresses, keeping a centralised record of all data. The data contained in the `Shared` struct can be seen in Listing 3.11.

```
1  struct _Shared {
2      Buffer * pMacKey;
3      Buffer * pEncKey;
4      Buffer * vMacKey;
5      Buffer * vEncKey;
6      Buffer * sharedKey;
7
8      Nonce * serviceNonce;
9      Nonce * picoNonce;
10
11     KeyPair * serviceIdentityKey;
12     KeyPair * serviceEphemeralKey;
13
14     //Fields I added required for client mode
15     KeyPair * picoIdentityKey;
16     KeyPair * picoEphemeralKey;
17
18     Buffer * extraData;
19 };
```

Listing 3.11: The `Shared` struct.

Each of the items of data in the struct have corresponding getters and setters, as well as functions used in the running of the protocol, such as a function to generate an EECDH shared secret using the data currently stored. Each type represents the following data:

- `Buffer` – A type from libPico that is essentially a length-prepended byte array. This is required as strings are usually represented as null terminated `const char *`s. This presents a problem when using DER encoded elliptic curve keys, as these may have null bytes and still be valid. Using a length prepended string avoids any problems associated with this. It also makes buffer overflow vulnerabilities less likely compared to a standard C `char*`.
- `Nonce` – A type from libPico, representing a cryptographic nonce with associated functions.
- `KeyPair` – A type from libPico combining together a public `EC_KEY` and a private `EVP_PKEY`
- `EC_KEY` and `EVP_PKEY` – Types from OpenSSL representing elliptic curve keys.

Each message type also has a struct in libPico that contains a pointer to the instance of the `shared` struct as well as any additional data that does not need to be shared with other functions, usually temporary values used when serializing and deserializing messages.

3.3.3 Start message

The start message is the first message sent in the SIGMA-I protocol. It is sent from the client to the service and is used to inform the service of the client's ephemeral key. This message contains:

- `picoEphemeralPublicKey` – the key used to generate a shared secret with the service using EECDH. The ephemeral keys are used instead of the identity keys to create forward secrecy. This is sent as a DER encoded elliptic curve public key, serialised to a Base64 string.
- `picoNonce` – a randomly generated nonce. This will be used to verify that the service is alive and performaing the protcol, and therefore further preventing any replay attacks. This is also sent as a Base64 encoded string.
- `picoVersion` – an integer representing the version of Pico being used. For my implementation this will always be 2.

To create this message I wrote the `messagestart_serialize` method, which takes a `messagestart` struct and a buffer as arguments. This function simply takes data from the `shared` pointed to by the `messagestart` and serialises it to a JSON dictionary. This dictionary is converted to a string and written to the buffer that can then be used to send out the message.

When the service receives this message, it performs EECDH to generate a shared key. The service then performs a key derivation procedure, creating five keys, an encryption key and a MAC key for both the service and client, and finally a shared symmetric key.

3.3.4 Service auth message

The service auth message is the first message sent from the service to the Pico client. This is the message that the service uses to authenticate itself to the client. It contains the following data:

- `serviceEphemPublicKey` – the service's EECDH public key.
- `serviceNonce` – a randomly generated nonce.
- `encryptedData` – a bundle of data encrypted using the service's encryption key.

- `iv` – the initial vector used in the encryption of the `encryptedData` field. An initial vector is some random string that is used in the cipher to change the initial internal state of the cipher. In effect, this means the same data encrypted with the same key but with different initial vectors will have different cipher text.

Since the `encryptedData` is encrypted, an eavesdropper will not be able to view its contents. `encryptedData` contains length prepended strings of the service's identity public key, a signature keyed with the service's identity public key and a MAC keyed with one of the keys derived from the EECDH shared secret.

The signature covers the service's ephemeral public key and the client's nonce. This proves that the original ephemeral key was generated by the service owning that specific identity key and the nonce confirms to the client that the signature was generated in response to its `startmessage`, preventing a replay attack.

The MAC covers the service's identity key. This is required to prevent certain identity misbinding attacks that would be possible without this proof. The MAC in Pico is a HMAC-SHA256 which is cryptographically secure. It is important that the key used in the encryption of `encryptedData` and the key used in the HMAC are computationally independent, i.e. neither gives information about the other, as if this is possible then either the HMAC or the encryption will be broken.

To support service auth messages in my implementation, I created a function `messageserviceauth_deserialize` that deserializes the message, then checks that all the data it contains is valid. This function can be seen in Listing 3.12. The first action taken is to deserialize the JSON of the message.

```
1      json = json_new();
2      result = json_deserialize_buffer(json, buffer);
3      servicePublicKeyBytes = buffer_new(0);
4
5      messageserviceauth->sessionId = json_get_decimal(json, "
       sessionId");
6
7      value = json_get_string(json, "iv");
8      base64_decode_string(value, messageserviceauth->iv);
9
10     value = json_get_string(json, "serviceEphemPublicKey");
11     shared_set_service_ephemeral_public_key(messageserviceauth->
       shared, cryptosupport_read_base64_string_public_key(value));
12
13
14     value = json_get_string(json, "encryptedData");
15     base64_decode_string(value, messageserviceauth->encryptedData);
16
17     value = json_get_string(json, "serviceNonce");
18     verifierNonce = shared_get_service_nonce(messageserviceauth->
       shared);
19     base64 = buffer_new(NONCE_DEFAULT_BYTES);
20     base64_decode_string(value, base64);
21     nonce_set_buffer(verifierNonce, base64);
```

Listing 3.12: Deserializing the `messageserviceauth` JSON dictionary.

Now that all of the data has been pulled out of the message, the shared secret must be generated and then the keys must be derived from that secret. Helpfully, libPico provides a function to do this for the service side, `shared_generate_shared_secrets` which I modified to create a new function, `shared_generate_shared_secrets_pico` which generates the shared secret on the client side. This works by using OpenSSL to generate the EECDH shared secret using the data in the `shared` struct, then pushing that through a key derivation function to create the necessary keys.

Once the keys have been created, I can decrypt the `encryptedData` field and parse the data from that. Decrypting `encryptedData` will give one long string that is a collection of length-prepended strings. Since the `buffer` type is essentially a length-prepended string, parsing requires passing the decrypted cleartext to a function `buffer_copy_lengthprepend`. This requires the byte offset of the cleartext to read from, and returns the number of bytes read. This means for multiple reads I can take the output of the previous read and pass that to the next one.

Now that all the data is parsed, the signature included can be checked by calling `messageserviceauth_verify_signature` from libPico, and the MAC can be checked by calculating what it should be and then comparing this with the MAC recieved in the message. My code for this can be seen in Listing 3.13.

```
1  result = messageserviceauth_verify_signature(messageserviceauth,
       messageserviceauth->signature);
2
3  if (result) {
4      mac = buffer_new(0);
5      vMacKey = shared_get_verifier_mac_key(messageserviceauth->shared
       );
6      serviceIdentityPubEncoded = buffer_new(0);
7      serviceIdentityPublicKey =
       shared_get_service_identity_public_key(messageserviceauth->shared
       );
8      cryptosupport_getpublicder(serviceIdentityPublicKey,
       serviceIdentityPubEncoded);
9      cryptosupport_generate_mac(vMacKey, serviceIdentityPubEncoded,
       mac);
10     buffer_delete(serviceIdentityPubEncoded);
11
12     result = buffer_equals(mac, messageserviceauth->mac);
13 }
14
15 return result;
```

Listing 3.13: Checking the signature and MAC.

3.3.5 Pico auth message

The Pico auth message is very similar to the service auth message except wherever the service would use its public keys, the client uses its pair. This means that the client's ephemeral public key is sent in plain-text, while its long term identity key, a signature over the two ephemeral keys and a MAC of its identity key are sent encrypted. This is how the client has more protection in the SIGMA-I protocol. Since the client initiated the protocol by sending the start message, it authenticates the service first. At this point, the client knows the identity of the service but not vice versa, so the client could choose not to reveal its identity if the service was not the service expected.

To implement this message, it required pulling together all of the data required, creating the signature and MAC and then serializing this into a JSON dictionary.

Since all of the keys needed have already been generated, the first step performed is to generate the signature. The libPico function `messagepicoauth_generate_signature` performs this and the result is passed to a buffer. Next the MAC is generated using the `picoMac` key. After these have been generated, the two output buffers are combined into a single buffer, and then the client's long term public key is appended to that. These are then encrypted using the `picoEnc` key with a new random initial vector. This encrypted data and the initial vector are then combined into a single buffer as a JSON dictionary.

3.3.6 Status message

At this point, if there were no errors in the protocol then both the service and the client have enough information to authenticate each other. In the case that something went wrong during the deserialisation of the Pico auth message at the service, the client would have no way of knowing. The status message fixes this, simply sending back a code to inform the client of the status of the protocol. This message can take two values;

- `MESSAGESTATUS_REJECTED` $= -1$
- `MESSAGESTATUS_OK_DONE` $= 0$

This message is encrypted so as to not leak anything to an eavesdropper. This encryption is done using the verifier's encryption key. The initial vector used in the cipher is appended to the message in plain text as before.

3.4 App structure

3.4.1 Database

During the processes of pairing and authenticating, data is required to be stored and read respectively. OpenSSL provides a function to do this by storing keys to plain text files, but to improve performance I used a database instead. This allowed me to store all of the data created by a pairing at once, and retrieve it simply using methods I created to wrap around the database.

For my database I used SQLite as this is bundled with iOS and is designed to work well on mobile devices that have power limitations. The interface included in iOS uses Objective-C, so to work more quickly I also used FMDB, which provides a Swift interface allowing more easy development with my Swift-based app.

First, I created a class that contains the data needed for a pairing, which can be seen in Listing 3.14.

```
1 class KeyPair {
2     let picoPublicKey: String
3     let picoPrivateKey: String
4     let serviceCommit: String
5     let extraData: String
6 }
```

Listing 3.14: The `KeyPair` class.

Whenever a key-pairing action is performed, a `KeyPair` object is created and filled with the relevent data. `extraData` is read from the initial key-pairing QR code and the `serviceCommit` is calculated by performing the SHA-256 hash algorithm on the service's public key presented in the QR code.

Getting the string representations of the Pico's public and private key are a little more difficult however. In memory, they are stored as DER encoded keys. As discussed before, a DER key could contain a null byte and still be valid which presents a problem when trying to get a string representation to store these keys.

To solve this, I first convert the DER encoded keys to PEM encoded keys. PEM encoding is more helpful as it is a Base64 representation of the key that is safe to be passed as a string. To do this, I used an OpenSSL type called `BIO`, which is an I/O abstraction. This is the function I mentioned earlier that is used to output a key to a file, but instead of using a file pointer as an argument I pass it a memory address. The `BIO` then stores the DER encoded key at that memory address. Now that I have a pointer to the key, I can create a buffer that wraps around that, encode it into Base64 and then use the inbuilt buffer functions to convert that to a string. Due to the interoperability of C and Swift, this string can be returned from a function and it will be a valid Swift string that can then be stored in the database.

Loading keys uses a similar process in reverse. The strings representing the public and private keys are retrieved from the database, decoded Base64 into a buffer, then a `BIO` is used to read the memory of the buffer. This is done for each of the keys, and finally the functions `d2i_PrivateKey_bio` and `d2i_EC_PUBKEY_bio` are used to convert the `BIO`s to private and public keys respectively.

3.4.2 TouchID

TouchID is a feature currently supported by most iPhones released in the past four years. It takes the form of a fingerprint scanner on the home button of the phone. This can be built into apps to provide greater security.

With the Pico app, a realistic attack vector would be to steal a user's phone while it is unlocked. In this scenario, the attacker would have access to every one of the user's services. To prevent this, for one of my extensions I implemented TouchID functionality into my app.

To do this, I used Apple's `LocalAuthentication` framework, which as the name suggests provides local authentication techniques, specifically in my case authenticating the user to the app using biometrics.

The first step is to create an `LAContext`, which is an object used to validate `LocalAuthentication` policies. Now the app can check whether the device that it is being run on actually supports TouchID by calling the `context.canEvaluatePolicy` method with the argument `LAPolicy.deviceOwnerAuthenticationWithBiometrics`. If this returns true, the TouchID process can be run.

To run TouchID, I used the `context.evaluatePolicy` method, with the following arguments

- `LAPolicy.deviceOwnerAuthenticationWithBiometrics` – Signals that the app wishes to authenticate using the fingerprint scanner.
- `localizedReason: reason` – A string that is displayed to the user stating the reason TouchID is requiring their fingerprint.
- `reply: { (success, error) -> Void in ...}` – A closure that will run when the TouchID process has completed. If the users is authenticated correctly, the code block will capture `success` and if it fails the code block will capture `error`.

What this means in practice is that I can just use the code straight from the code handler in the closure block contained in an `if (success)` statement and error handling can be done with an `if (error)` statement.

The only problem this causes is an issue with threading. To display message boxes to the user when the SIGMA-I protocol completes, I pass the code handler a pointer to the main `ViewController` of the app, which allows me to access methods from the view controller inside the code handler. Inside the `UIViewController`, I have a method called `displayMessage` that takes two strings as arguments, a title and a message, and displays a pop up box on the phone screen displaying the title and message.

When authentication via TouchID is requested, a new thread is created for this to run on. This is so that background processes can still continue while the fingerprint is being scanned, although in my app I blocked all other processes as to not repeatedly scan the QR code and start the protocol many times. This means that the majority of the code handler code runs in a secondary thread, however, iOS mandates that all UI changes must be done through the main thread. If they are not, the app hangs for around 15 seconds or until the screen is tapped. To fix this problem, I force the UI updates to run in the main thread. This is done by wrapping any UI code inside

```
1 DispatchQueue.main.async {
2     //Run code here
3 }
```

3.5 Summary

This concludes the implementation of my Pico client. I have built and integrated all of the compenents necessary for my app to function with the Pico system and I have also explored how extra security could be added with biometrics using TouchID. During the course of my development I created the SwiftECC library to perform ECC operations purely in Swift.

Chapter 4

Evaluation

This chapter will evaluate my project, taking into account my original success criteria and how it performs. I also explore how the choice of elliptic curve affects the usability and security of Pico, and if the current curve choice could be improved.

4.1 Success criteria

4.1.1 Functionality

Reasoning

Whether or not my app works with the existing Pico system is arguably the most important part of my project. My original success criterion for functionality was for the app to be compatible with the current Pico web system. Early on in the project, it became clear that this would require a large amount of 'grunt' work, so instead I decided to focus on getting my app to work with the computer log-in system, replacing passwords for Windows and Linux systems with Pico.

Methodology

To test whether my app worked correctly with the Pico system, I had to test my app on one of the machines in the Computer Laboratory that has the complete Pico environment set up. On these machines, I could pair my app and then authenticate to log into them instead of providing a username and password. Since this is a test with a binary result, either it works or it does not, I simply attempted to pair my phone to a machine using my app, and then attempted to log in to that machine using my app.

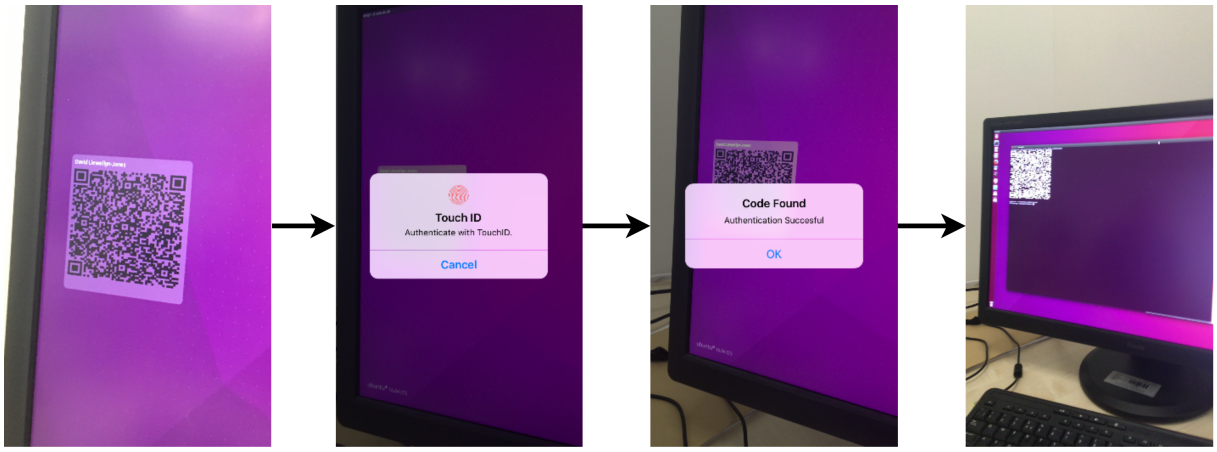

Figure 4.1: Screenshots showing the stages of authenticating on a machine; code found, TouchID, authenticated.

Results

This test was a success: my app successfully paired and authenticated with the desktop pico software, allowing me to log in. This process is shown in Figure 4.1. Due to this success, I could move on to testing how my app compares to the current Pico implementation on Android.

4.1.2 Speed

Reasoning

To ensure that the app is useful, it must be no more inconvenient for users than a username and password system. To this end, it is critically important for the app to complete both pairing and authentication quickly with minimal effort.

Methodology

To record the time it took for my app, and the current Android implementation, to pair and authenticate I modified the `test-qr` program to record and print out timing data each time it was run. This timer starts the moment the program is run and stops when the SIGMA-I protocol completes. To ensure all tests were run fairly I opened the app on the phone and pointed the camera at the area of my screen that the test app would open in and then ran `test-qr`. Since the timer would start as soon as the program would run, and therefore the same time the QR code appeared on screen, the timing result would be for the whole Pico process, from scanning the QR code to finishing the protocol. I used the same strategy for testing both my implementation and the Android implementation. I then did 50 pairings and 50 authentications for each test case and calculated the mean time for each.

One area that created an issue for a like for like comparison was the fact that my implementation uses TouchID for extra security, while the Android implementation does not. To gain a more accurate insight into my app, I ran the tests twice, once with TouchID enabled and once with it disabled. To ensure consitency, I ran all of the TouchID tests with my thumb placed on the fingerprint scanner before the test started and kept it there throughout. This meant that TouchID would authenticate as soon as it was able to and removed any human errors that could be introduced when placing a finger on the scanner.

For all of my tests, `test-qr` was run on a late 2014 Macbook Pro (2.6 GHz i5 with 8GB of RAM), my app was run on a iPhone 6s and the Android implementation was run on a Nexus 5.

Results

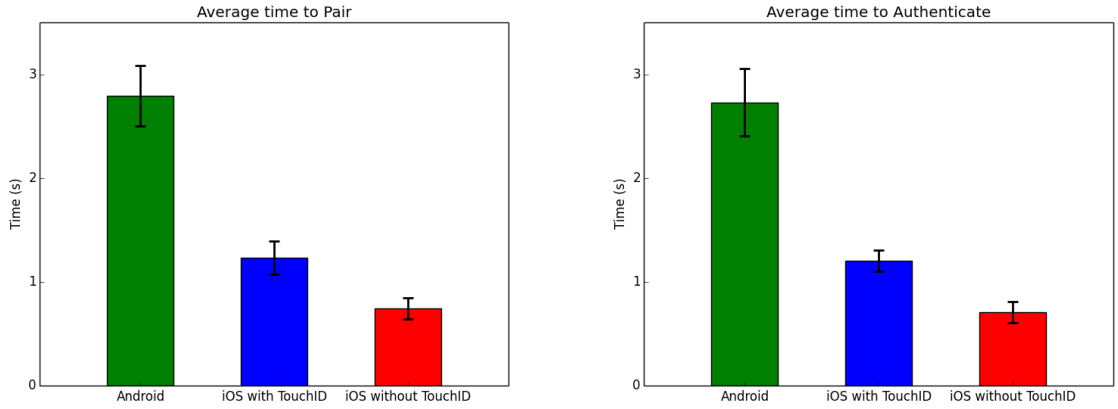

Figure 4.2: Average time to pair and authenticate using the secp192r1 curve on the old Pico implementation and my new implementation with and without TouchID.

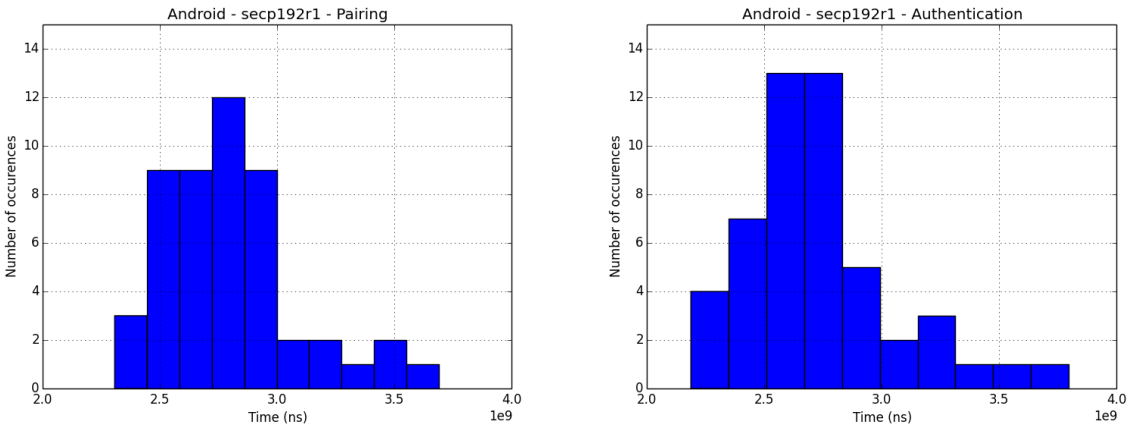

Figure 4.3: Distribution of timings of the Android app with the secp192r1 curve.

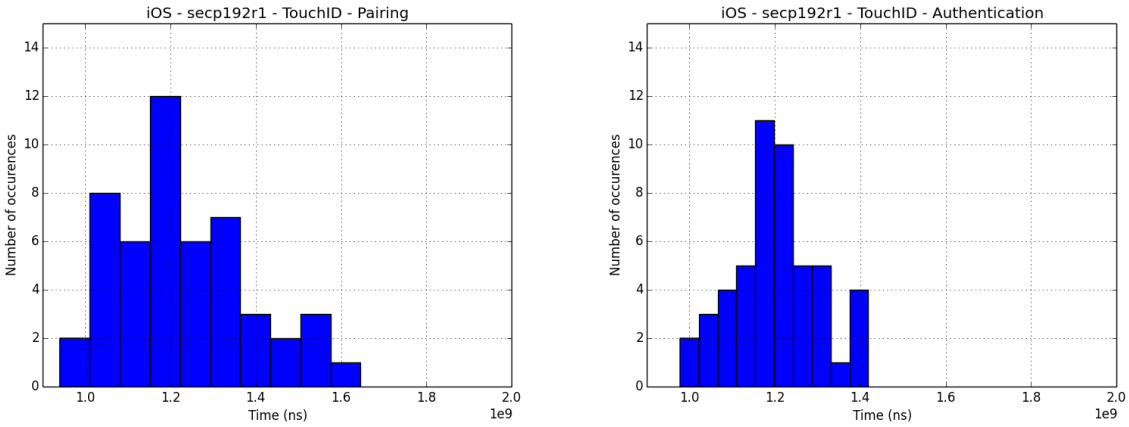

Figure 4.4: Distribution of timings of my iOS app with the secp192r1 curve and with TouchID enabled.

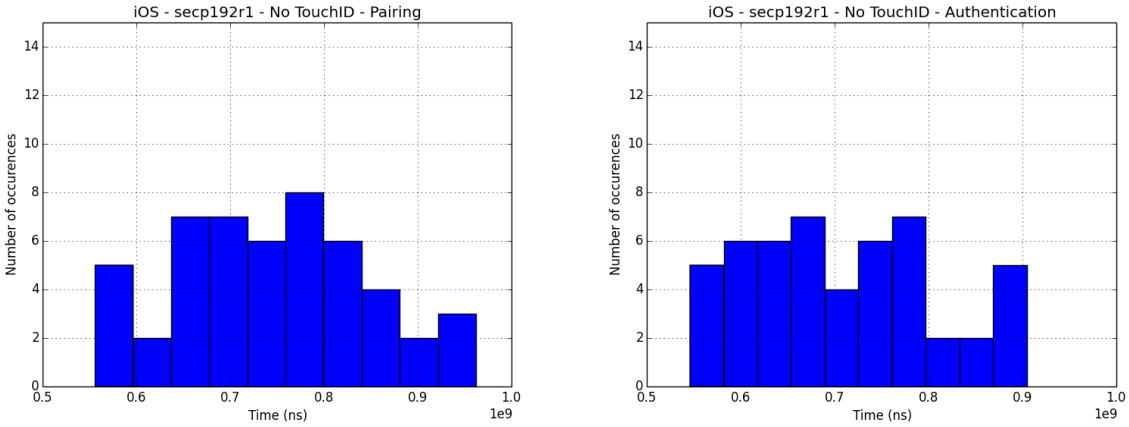

Figure 4.5: Distribution of timings of my iOS app with the secp192r1 curve and with TouchID disabled.

Comparing my app and the Pico team's implementation in a like for like situation with no TouchID, there is a large improvement in speed in my implementation that can been seen in Figure 4.2. The average authentication time was 3.85 times faster and the average pairing time was 3.72 times faster. The distribution of times for these tests can be seen in Figures 4.3, 4.4 and 4.5. I propose three reasons for this dramatic increase:

- **Streamlined workflow** – The Java app has a very complex internal structure that is, in my opinion, needlessly complex. For my implementation I focused on having a streamlined workflow that could run as efficiently as possible.

- **Language choice** – The core functionality of my app is written in C, while the core of the Android app is written in Java. In general, C runs much faster than Java due to its less complex nature and the fact that Java runs in a virtual machine while C is compiled to assembly code [10]. The wrapper of my app was written in Swift, which has also been shown to be faster than Java in most cases [11].
- **Hardware differences** – The Android app was tested on a phone slightly less than two years older than the phone my app was tested on. Even though on paper, the Nexus 5 has similar specifications to the iPhone 6s, there are likely to be some optimisations present on the iPhone that allows it to run faster.

In practice, it is likely a combination of the three that gives my app such better performance. An interesting thing to note is that my app took, on average, 6% longer to pair than to authenticate, while the Android app only took 2% longer. This is likely due to the increased work in the code handler for pairing rather than authentication. This lends credit to my idea that my app is more streamlined. Since the same amount of extra work has to be completed for both apps, if my advantage was due to better hardware, the extra time taken would be expected to be a linear percentage on both devices. It is not, in fact the actual increase in time is similar across both devices, 0.04 seconds with my app vs 0.06 seconds on the Android app. The small difference is likely to come from working with the database, as my app uses a single SQL query to insert whereas the Android app uses an Object-Relational Mapper to insert objects into the database, which will add some time.

When comparing my app with TouchID *vs* the Android app without, there is still a clear increase in speed in my app. Adding TouchID means that for authentication, my app is 2.28 times faster and pairing is 2.27 times faster. This is an impressive result given the added security.

The standard deviation for my app with and without TouchID was only 1% different, which is expected as the only difference should be the near constant time used to authenticate with TouchID.

4.1.3 App Store guidelines

Reasoning

Since my app is a project with a real-life use case, the ultimate aim is to have users running my Pico app on their phones. On iOS, the only way for users to install apps is through Apple's App Store. For an app to be published on the App Store, it must meet certain requirements. I must make sure that my app meets those requirements.

Methodology

To check my app meets the guidelines, I went through the list available on Apple's website [12] and evaluated whether or not my app met each specific guideline.

My initial idea was to submit the app and see whether it was accepted by Apple, but I did not for two reasons:

- Submitting an app requires a full developer account, for which payment is required.
- To submit an app it must be a full version with all back-end services working. Since the back end code for Pico has not been released into the public domain yet, this would stop the app from being accepted.

Results

After checking through the guidelines, it is clear that my app would meet all of them, apart from guideline 2.1, which states that all app back-end services should be functional before an app can be released on the app store. Since the Pico team control the release of the back-end Pico code, I have no control over this point. The code is due to be released in June however, and at that point my app will meet all of the guidelines. I consider to have met the success criterion in this regard, as publishing the app would be pointless without the back-end services running anyway.

4.2 Curve choice

4.2.1 Curve length

Reasoning

The length of a curve is the number of bits in the prime of the field of the curve. In general, a higher length leads to more security but more complex computations for the functions on the curve. The curve that the Pico system currently uses is the secp192r1 curve. This is a relatively small 192-bit curve that was chosen due to its quick run time and small key size that reduces the amount of data sent over the networked compared the larger curves. NIST currently recommends a minimum curve size of 224 bits [13], but the greater the size, the more secure the system will be. Due to the increase in speed my app has achieved, it is a good idea to explore longer curves and see if their usage has any dramatic effect on run time.

Methodology

I tested three curves, secp192r1, secp256r1 and secp512r1, with lengths of 192, 256 and 512 bits respectively. I used my data from the previous test for the secp192 curve and for the other curves I repeated the same process as testing the app in the previous section, recording times for 50 pairings and 50 authentications for each curve. I enabled TouchID for all of these tests as the earlier results had made it clear that even with TouchID enabled, the app still ran at a very usable speed with the smallest curve. Also, an attacker stealing a user's phone is a much more likely attack than breaking the curve, at least for the next few years.

Results

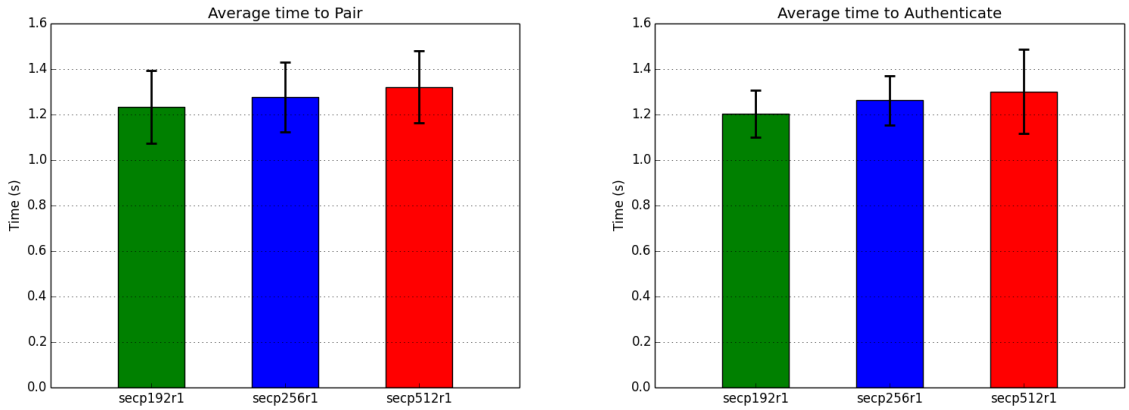

Figure 4.6: Average time to pair and authenticate using the secp192r1, secp256r1 and secp512r1 curves with my app using TouchID.

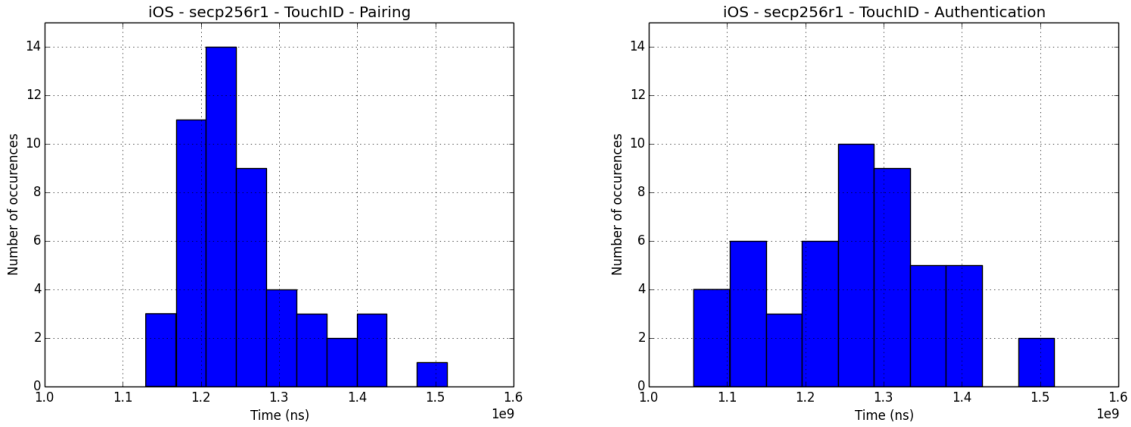

Figure 4.7: Distribution of timings of my iOS app with the secp256r1 curve and with TouchID enabled.

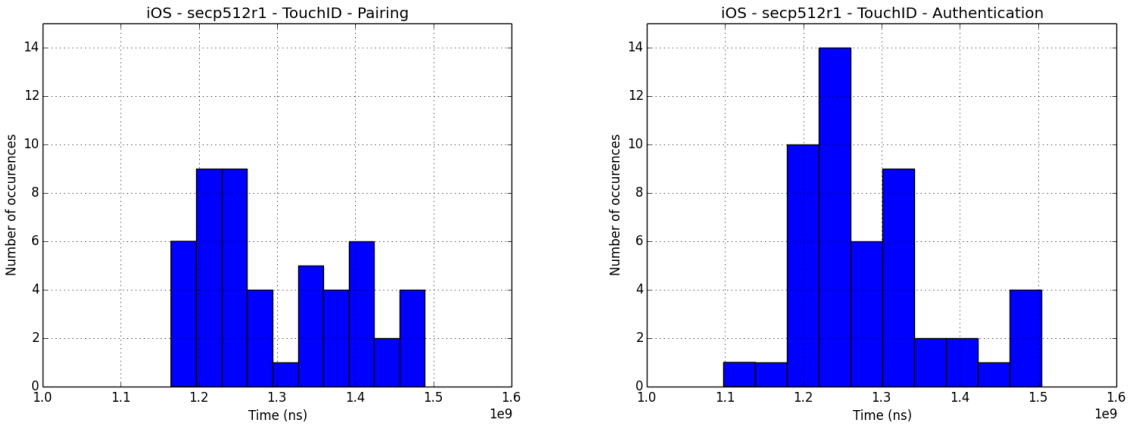

Figure 4.8: Distribution of timings of my iOS app with the secp512r1 curve and with TouchID enabled.

As before the average times are presented in Figure 4.6 and the distributions are shown in Figures, 4.7 and 4.8. I have ommitted duplicating the histogram for the secp192r1 curve. These tests showed very positive results, for the secp256r1 curve, the average pairing time was only 4.1% slower compared to the baseline secp192r1 while authentication was 2.5% slower. For the even larger secp512r1 curve the average pairing time was 7.3% slower then secp192r1 and authentication was 8.3% slower. On numbers of this order of magnitude, the largest difference of 8.3% translates to around 0.1 seconds.

4.2.2 Random vs Koblitz curves

Reasoning

There are two main types of curves used in most cryptosystems, random and Koblitz curves. With a random curve, the parameters are chosen psuedo-randomly in an attempt to make it such that if one curve is broken, other curves will not be as there are no links in their parameters. Koblitz curves have their parameters chosen relatively rigidly and allow for more efficient operations [14]. It would be useful to test the difference between a random curve and Koblitz curve of the same length, to see whether or not the improved performance was worthwhile.

Methodology

I tested these curves in the same way as I tested curves of different bit lengths. To make a comparison I tested secp192r1, a random curve, against secp192k1, a Koblitz curve.

Results

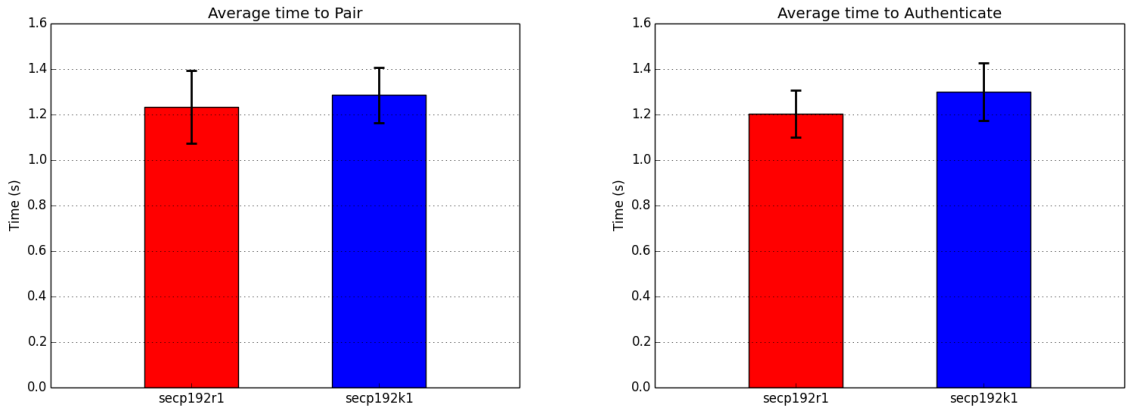

Figure 4.9: Average time to pair and authenticate using the secp192r1 and secp192k1 curves with my app using TouchID.

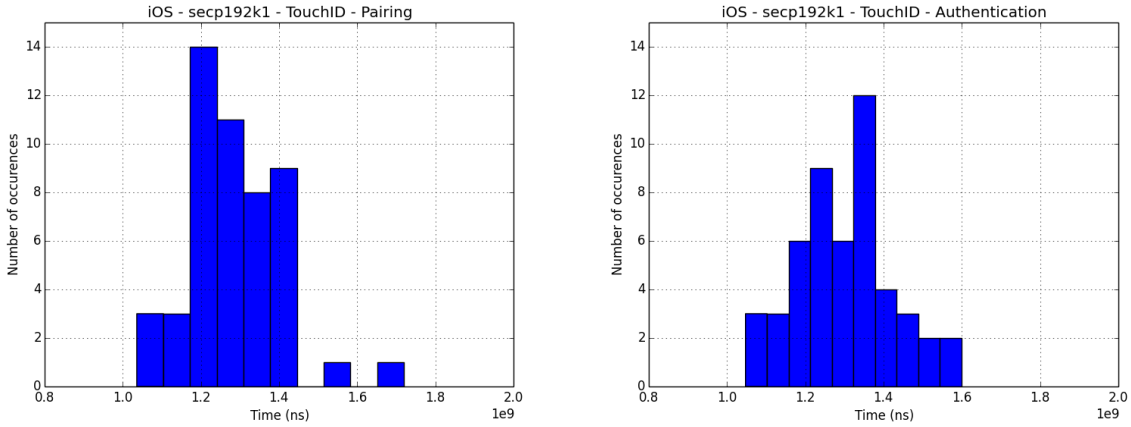

Figure 4.10: Distribution of timings of my iOS app with the secp192k1 curve and with TouchID enabled.

The average times are shown in Figure 4.9 and the distribution of times on the Koblitz curve are shown in Figure 4.10. Unexpectedly, in my tests the Koblitz curve actually turned out to be slower than the random curve. Pairing was 4.9% slower and authentication was 8.3% slower, taking a similar time as the much stronger secp512r1 curve. An interesting point is that the Koblitz curve is the only curve for which it took longer to authenticate than to pair. This leads me to believe the reason the curve ran slower was due to a lack of optimisation for Koblitz curves in both my app and the Pico test app, as in theory the curve should be more efficient.

There are some important factors to take into account when choosing between random or Koblitz curves however. Koblitz curves are a few bits weaker than a random curve of the same length (although this doesn't pose a major threat) but more importantly, since Koblitz curves are all built in a similar way, it is possible that if one gets broken then all of them will. Random curves do not face this problem as each has its parameters chosen independently.

On the other hand, since Koblitz curves are built in a way that is known by the public, they are known to not have backdoors. Random curves have no such guarantees, and recently there have been reports that the NSA generated some random curves that have backdoors in them, although this is unconfirmed [15]. A backdoor in the elliptic curve would allow anyone with knowledge of that backdoor to break the keys and therefore render Pico useless.

4.3 Summary

From my results, it is clear to see that my app meets my initial success criteria and outperforms the expected level of speed. With this in mind, I believe it would be beneficial to change the curve used by the Pico system to the secp512r1 curve, which has a very high level of security while only marginally decreasing usability by slightly increasing the time taken to pair and authenticate. I would also strongly recommend using TouchID whenever possible. Since there are no known attacks on the SIGMA-I protocol, the biggest risk is gaining physical access to a victim's phone. Using TouchID would provide a slight decrease in usability due to a small increase in time to use Pico and having to use the fingerprint scanner, but provides a much greater level of security for the user.

Chapter 5

Conclusion

This chapter concludes my dissertation that has covered the design, implementation and evaluation of my Pico client for iOS. Overall, it has been a fun experience and has greatly increased my knowledge of cryptosytems and their implementations. There were some challenges, such as working on a project with a large existing codebase with little documentation and Apple's rather arbitrary rules for development and forced updates, but overall it has been a good experience.

5.1 Achievements

My project overall has been a success, meeting all of my success criteria (with the small exception of focusing on computer log-ins rather than website log-ins) and performing better than the Pico team's current implementation of the Pico app on Android. My app allows users to securely pair and authenticate with their machines without having to use passwords or worry about carrying any hardware they don't already carry.

In addition to the core project, I researched ways in which Pico could be improved. From my findings I implemented TouchID, combating the most plausible attack an adversary could execute. I also recommended changing the underlying elliptic curve used by Pico, by comparing different lengths and constructions to find the best curve for Pico.

I also implemented an ECC library in Swift, which as far as I can tell is the first library of its kind to be written purely in Swift. There is work that could be done to improve this but overall it works as expected and provides all of the functions necessary to implement a key exchange protocol.

5.2 Lessons learned

The project has been a great learning experience for me, easily being the largest single project I have worked on. My main take-away has been the importance of creating and

sticking to a detailed plan. A lot of time was wasted during the early stages of my project as I didn't have a clear view of what to do or how to make progress.

Another very important lesson I learnt was to meet with members of the Pico team as much as possible. As mentioned earlier, Pico has very little publicly accessible documentation, so I found the best way to gain an understanding of how the system works was to ask people directly involved with the Pico project.

On a more technical note, I learnt to be a lot more thoughtful of technical limitations while planning on how to implement a feature. For example, C's lack of objects took a while for me to get used to as most of my experience in the past has been with object-oriented languages. I also learnt to be wary of using languages that are still in development. A major version of Swift was released around a month prior to beginning my project that had introduced many fundamental changes to the language, such as removing regular `for` loops. This meant a large amount of the literature available was outdated, which meant a lot of my work required some trial and error.

5.3 Further work

There are many avenues that could be explored involving my app and the Pico system, but there are a few that stand out as the most important:

- **Bluetooth** – Supporting Bluetooth channels would greatly increase the usability of my app, allowing users to authenticate without scanning a QR code. This would have been a nice feature to implement but the Bluetooth capabilities of iOS are very difficult to use when connecting to a non-OSX platform. Since the current Pico system only works on Linux and Windows, this would have been a very difficult task.

- **Cloud Backup** – It would be useful to be able to backup pairings to some cloud service so if a user lost access to their phone, they could install the app on a new phone and continue as normal. With my current implementation it would be possible for a power user to backup their pairings through XCode by saving the database file created by the app.

- **Swift ECC Improvements** – My Swift elliptic curve cryptography library has lots of room for improvement. Writing a new big Int library from scratch would probably be the best way to do this, as the library I currently use has very inefficient modulus and divide operations. Writing an ASN1 compiler/decompiler would be beneficial as it would allow my library to plug into existing cryptosystems, including Pico.

- **Pico Lens compatibility** – My original success criteria was for my app to function with Pico Lens, which allows users to us Pico to authenticate to websites. Adding this functionality would greatly increase the usefulness of Pico for users.

Bluetooth support, cloud backup and Pico Lens functionality have been implemented on the Android app, so these would be the most important features to add to make my iOS app equal to the Android app.

I am giving my source code to the Pico team to be released as open source under the BSD license with the hope that these features will be implemented.

Bibliography

[1] T.Moore and R. Clayton: *"An Empirical Analysis of the Current State of Phishing Attack and Defence"*.
http://www.cl.cam.ac.uk/~rnc1/weis07-phishing.pdf

[2] Andy Greenberg: *"The NSA confirms it: Russia hacked French election infrastructure"*
https://www.wired.com/2017/05/nsa-director-confirms-russia-hacked-french-election-infrastructure/

[3] Anne Adams and Martina Angela Sasse: *"Users are not the enemy"*.
http://discovery.ucl.ac.uk/20247/2/CACM%20FINAL.pdf

[4] Frank Stajano: *"Pico: No more passwords!"*
http://www.cl.cam.ac.uk/~fms27/papers/2011-Stajano-pico.pdf

[5] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot and Frank Stajano: *"The quest to replace passwords: a framework for comparative evaluation of Web authentication schemes"*.
https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-817.pdf

[6] https://github.com/AdamRoberts96/PicoiOS

[7] Apple: *"AV Foundation"*
https://developer.apple.com/av-foundation/

[8] Kevin Parrish: *"Fail0verflow Obtains PS3 Cryptography Key"*.
http://www.tomsguide.com/us/PlayStation-Console-Private-Cryptography-Key-fail0verflow-linux,news-9542.html

[9] Hugo Krawczyk: *"SIGMA: the SIGn-and-MAc Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols"*.
http://webee.technion.ac.il/~hugo/sigma-pdf.pdf

[10] Sebastian Nanz and Carlo A. Furia: *"A Comparative Study of Programming Languages in Rosetta Code"*.
https://arxiv.org/pdf/1409.0252.pdf

[11] *"Swift versus Java by benchmark task performance"*.
https://benchmarksgame.alioth.debian.org/u64q/swift.html

[12] Apple: *"App Store Review Guidelines"*.
https://developer.apple.com/app-store/review/guidelines/

[13] Elaine Barker: *"Recommendation for Key Management"* .
http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

[14] Kristian Bjoernsen: *"Koblitz Curves and its practical uses in Bitcoin security"*.
https://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Bjoernsen.pdf

[15] Daniel J. Bernstein and Tanja Lange: *"Security dangers of the NIST curves"*.
http://www.hyperelliptic.org/tanja/vortraege/20130531.pdf

Appendix A

Project Proposal

Adam Roberts
St John's College
ar752

Part II Project Proposal

Pico for iOS

20th October 2016

Project Originator: Dr. Frank Stajano

Resources Required: See Resources Declaration

Project Supervisors: Dr. David Llewellyn-Jones and Mr. Seb Aebischer

Director of Studies: Dr. Robert Mullins

Overseers: Dr. Ian Wassell and Prof. Larry Paulson

Introduction

Pico is a project from a team in the lab that aims to make passwords obsolete. Currently, most people follow one of two main choices for manging their passwords.

The first choice is to create an easy-to-remember password. This leads to passwords being easily broken as humans tend to be quite predictable, also people will tend to use similar (or in the worst case, the same) passwords across multiple accounts, meaning if one account gets compromised then a potential adversary could have access to all of that persons accounts. The second choice is to have a long, hard-to-remember password. This would be harder to crack but now people resort to using password management software meaning that if someone gains access to a person's machine, they also gain access to all of their accounts. All passwords are vulnerable to less technical threats too, such as phishing.

Neither of these approaches are perfect, so Pico tries to do this by removing passwords altogether and using a portable device for authentication. The current implementation of Pico works by displaying a QR code when a user attempts to log into a website and then using a mobile application to scan this code to authenticate the user and log them in. My project would be to build an iOS app with this functionality.

I have no experience with iOS development, so there will be a lot of time at the beginning of the project researching Apples development framework and a language suitable for iOS development. I will likely use Swift as this is the language Apple seems to be supporting going forward.

Starting Point

There is currently no iOS app for Pico, but there is an Android app which can be used as a reference. I will have to implement some of the Pico libraries on iOS, this will either be by re-writing them or by using a wrapper. I will test these approaches to see which is more suitable. The current server-side implementation works with the Android app so I will build around that, but it may need to be changed to fit iOS-specific requirements as the need arises.

I have some mobile development experience gained from developing an Android app as part of my Part IB Group project and I have a solid grounding in security from the Part IB security course.

Substance and Structure

The main challenges in this project will be to implement the protocols used by Pico on a new platform. Pico uses public-private keys to authenticate, so I would have to implement this system natively on iOS. This will be the biggest technical challenge as I

will have to implement any algorithms in a way which the performance is fast enough to not inconvenience users and also in a way which is secure.

There are three types of QR code that the app would have to be able to deal with:

- Blue codes, which link a user's Pico account to other online accounts.
- Red codes, which link a user's Pico account to a specific browser on a specific machine.
- Green codes, which allow a user to log in to a service once their machine and accounts are linked.

Each of these codes will require separate modules to deal with. Each will require certain processing on the phone and then communication with a Pico server, either natively implemented in a service or via a browser plug-in.

Ensuring that all of this computation is done securely and quickly will be the core work of the project. This is important as Pico must be both more secure than passwords and no more inconvenient for people to use it instead.

Possible extensions could include:

- Bluetooth connectivity and continuous authentication. This would make the user experience more convenient as a user would not have to scan a QR code but rather Pico can detect that their phone is nearby.
- Backup and restore to cloud services, so that if a user loses their phone they can restore their accounts to a new phone.
- Enhanced iOS-specific security features such as Touch ID.

Criterion For Success

The criterion for success for this project would be to:

- Create a functioning application which can handle all basic Pico functionality: linking web service accounts to a Pico account, linking Pico accounts to a browser and allowing users to log into web services using Pico.
- Handle all computation in a similar time frame as the current Android implementation, ideally faster than a user can log in with a standard username and password.
- Fulfill the Apple App Store guidelines, as although I will not be releasing the app it may be released in the future.

Plan

Weeks 1-2 — 10/10/16 - 23/10/16 — Michaelmas Term

- Research iOS development and create a Hello World! application. This would include getting familiar with one of the languages used in iOS development, probably Swift.
- *Submit project proposal.*

Weeks 3-4 — 24/10/16 - 6/11/16 — Michaelmas Term

- Continue researching iOS development and implement a QR Reader using Apples AVFoundation framework.
- *At this point I should have a basic application which can be used as a framework for more development.*

Weeks 5-6 — 7/11/16 - 20/11/16 — Michaelmas Term

- Implement the code scanning feature of Pico, so the app can read a code and get its data and what action should be performed.

Weeks 7-8 — 21/11/16 - 4/12/16 — Michaelmas Term

- Implement the network protocol so that the app can communicate with the browser the user is currently using to try and log in.

Weeks 9-10 — 5/12/16 - 18/12/16 — Christmas Vacation

- Implement the database connectivity in Pico so that the app can access login credentials and provide them to the browser.
- *At this point I will have a functioning application which I will release to my Supervisors*

Weeks 11-12 — 19/12/16 - 1/1/17 — Christmas Vacation

- Work on the UI of the app to provide a better user experience.
- Perform automated and real world testing, fixing any problems found along the way.

2/1/17 - 29/1/17 — Christmas Vacation and Lent Term

- Contingency time set aside for any major issues found up to this point.

Weeks 13-14 — 30/1/17 - 12/2/17 — Lent Term

- Evaluate the project, recording data from testing and usage of the application.
- *Submit Progress Report.*

Weeks 15-16 — 13/2/17 - 26/2/17 — Lent Term

- Create a first draft of the dissertation and give this to my DoS and Supervisors for feedback.

Weeks 17-18 — 27/2/17 - 12/3/17 — Lent Term

- Create a second draft of my dissertation, this should be similar to the final version with the only major differences being wording and style.

Weeks 19-20 — 13/3/17 - 26/3/17 — Lent Term and Easter Vacation

- Receive feedback from my DoS and Supervisors about my dissertation.
- Complete the final version of the dissertation and submit it well before the deadline to avoid any issues in printing or sending.

Resources Declaration

I will be using my personal 2014 Macbook Pro and iPhone 6s for development and testing. I will keep backups locally, through online services and on the MCS machines. In case of a fault with my personal laptop there are Macs with Xcode installed in my college library which can be used for development and I have access to other iPhones if the need arises.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

I will require access to the existing Pico for Android code and other parts of the current Pico codebase.

I pledge to release all source code and documentation (dissertation included) as open source under the BSD 3-clause licence, which is a condition for me being accepted as a project student by the Pico project.