Biometric Bit locker

Design Document

Team: 23

Client/Advisor: Akhilesh Tyagi

Yousef Al-Absi/DevOps, Cole Alward/Scrum Board Master, Morgan Anderson /Scribe, Larisa Andrews/Scrum Master, Ammar Khan/Product Owner, Justin Kuhn/Testing Engineer

sdmay19-23@iastate.edu

http://sdmay19-23.sd.ece.iastate.edu/

Revised: 12/01/18

Contents

Figures iii
1 Introduction 1
1.1 Problem Statement 1
1.2 Purpose 1
1.3 Intended User and Users 1
1.4 Assumptions and Limitations 1
1.5 Project Goals and Deliverables 2
2 Design Specifications 3
2.1 Proposed Design 4
2.2 Design Analysis 7
3 Testing and Implementation 7
3.1 Process Details 7
3.2 Interface Specifications 9
3.3 Hardware/Software 9
3.4 Functional Testing 9
3.5 Non-functional Testing 11
3.6 Modeling and Simulation 13
3.7 Implementation Issues and Challenges 14
4 Closing 15
4.1 Conclusion 15
References 16

Figures

Figure 1 Initial Design Overview 1
Figure 2 Statistical Concentration/Correction for a Profiled User 6
Figure 3 Statistical Concentration/Correction for a User other than Profiled User 6
Figure 4 Process Flow Diagram 8
Figure 5 Model of layout 14
Figure 6 Simulation of trace 14
Figure 7 Simulation results 14

1 Introduction

1.1 Problem Statement

Asymmetric encryption is an encryption that allows us to hide our data and makes sure it's secure. Typically, files would be encoded using a public key and then they would be decoded using a private key. The private key is stored in the TPM (Trusted Platform Module) which is a cryptographic module that enhances computer security and privacy. TPM chips are usually discrete chips soldered into a computer's motherboard which allows them to be separate from the rest of the system. Android phones, however, lack the TPM chip. Therefore, encryption keys must be stored on the device somehow. If the keys are stored on the devices, they can be found and could fall into malicious hands.

1.2 Purpose

Our solution to the problem is to dynamically generate the key using a PUF. If we do that, the key will not be stored anywhere, and it can only be generated at runtime. Solving both the security issue and the storage of the key. With this key, we will de-encrypt the data stored in the phone. This is the purpose of our project. To use the given library to generate a private key dynamically and encrypt and de-encrypt data on the user's phone.

1.3 Intended User and Users

The integrated PUF would be used by any person who has a phone with information that they deem worthy of protecting.

Currently according to Statista [1], 54.1% of people in the united states are currently using an Android. So, we would want to reach that whole market with our application. Since nowadays most people with a phone have data worth encrypting.

1.4 Assumptions and Limitations

Assumptions:

- The PUF library is working at the beginning of the second semester.

  We have found several issues with the library that was given to us. Our hope to fix the issues we have found in the library by the beginning of next semester. So that we can use the working PUF to be able to develop our application.

- Android continues to support full disk encryption.

Android currently supports full disk encryption. Although, because of the legality that comes with encrypting the full disk, android is thinking of not supporting it.

- Android allows developers to integrate in the boot sector.

  There is a chance that android will not allow us to integrate our application into the boot sector. If this is the case than the application will not reach its full potential and will have to just be an application.

Limitations:

- Nexus 7 hardware.

  The Nexus 7 is the hardware that school provides, and what the PUF was designed on. The team decided it would be best to continue implementation on this hardware. Although, this will provide some limitations. Nexus 7 is currently aiming API 23. Which is much lower than the current android version. This prevents us from using some features that may be used in newer versions.

- Previous PUF library implementation and research.

  We must use the previous implementation of the PUF. Whether we agree with the research or not it is a part of project to use it.

1.5 Project Goals and Deliverables

Goals:

- To continue development on the given open source PUF library.
- To make the PUF work as a lock screen by asking a user to draw shapes to unlock the phone and authenticate then properly.

Deliverables:

- A well tested PUF Java gestures library
  - An open source library that should be released with unit tests and rewritten methods. We need to provide a valid testing framework and an appropriate architecture for the software.
- A PUF based Android app

- The app should act as a lock screen whenever the phone is closed, it'll authenticate users by asking them to draw a shape and it should only work for the correct user. The app will also automatically start when the phone boots.

2 Design Specifications

As the PUF is already given to us, we're not designing anything new for it but rather only fixing it to make sure that it meets the requirements listed below. We're also designing an android phone application.

Functional Requirements:

- Application should appear whenever the phone is locked.
- Application should be able to create multiple profiles.
- Application should be able to authenticate users.
- Application cannot be closed when its locking phone.
- PUF should encrypt and decrypt user profiles.

Non-Functional Requirements:

- Performance: Response time for authentication should be less than 4 seconds.
- Scalability: Application should have more than 2 profiles.
- Maintainability: The repository should update the application automatically
- Security: Only the proper user can unlock the application.
- Data Integrity: Data will be encrypted and decrypted successfully provided the correct key.
- PUF should have an accuracy of at least 80%.

2.1 Proposed Design

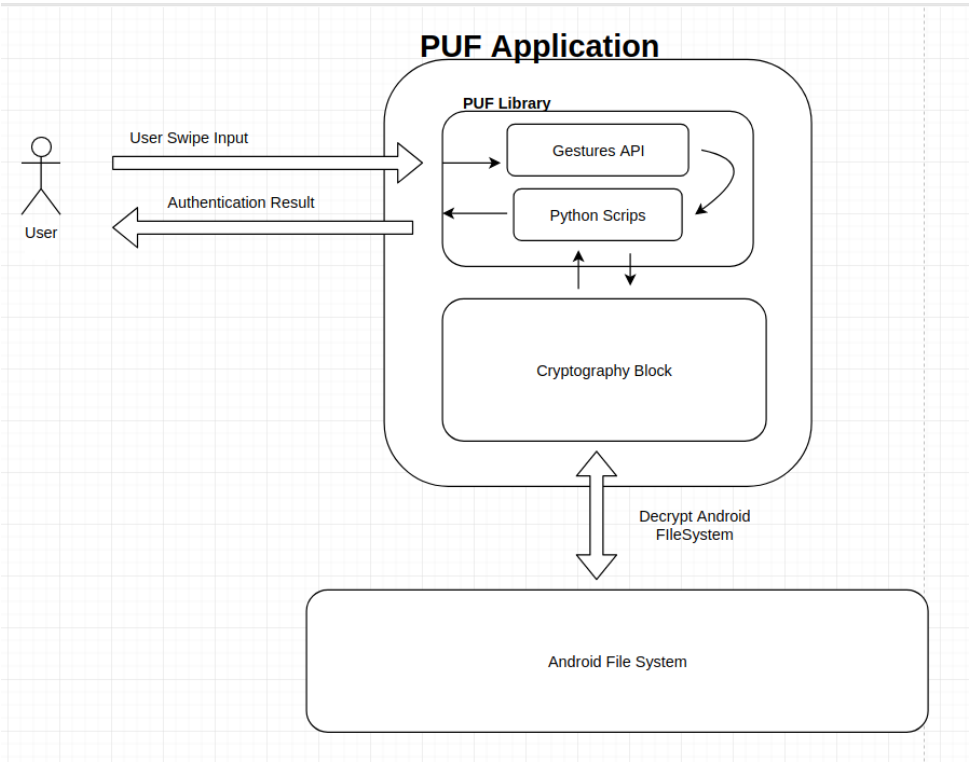

Figure 1 Initial Design Overview

For our project, some of the groundwork for the solution has already been completed. The current solution is to develop an Android application that uses data generated from pressure readings returned by the screen when the user traces a generated shape like the native Android unlock pattern. The data would be generated by a Physical Unclonable Function (PUF), which has already been implemented. As stated earlier, mobile devices lack the Trusted Platform Module (TPM) chip which can be found on most laptop computers. This enables full disk encryption on laptop computers and each TPM has its own unique and secret RSA key that will decrypt the disk. Since each chip has its own unique RSA key, this makes it an extremely secure method for encryption. With the lack of this chip on mobile devices, there is a lack of such a secure method for encryption. Ideally, the PUF will emulate the security of the TPM chip by dynamically generating the key every time and thus getting rid of the need to save it. The key generation will occur when the user traces the pattern generated by the app. The app provides the user with a given trace and has them trace it a certain amount of times depending on the selected user selected strength (higher the strength, the higher the number of traces) to develop a profile for the user based on the patterns for pressure and speed the user exhibits for that particular pattern. Furthermore, the hardware for no two devices is the same, even among

devices of the same model. Therefore, the pressure readings will vary from screen to screen and thus importing a profile to another device should result in failure even if the same person traces the pattern on the new device. Therefore, this implementation will lead to both user authentication and device authentication, thus leading to maximum possible security with the given hardware.

PUF Library Block

The pattern tracing and user authentication features are all encapsulated within the PUF Library sub block in **Error! Not a valid bookmark self-reference.**. These features have all been implemented but need to be updated, reworked and thoroughly tested before we can deem them functional or reliable. The implementation of the pattern tracing and user authentication system currently has the user trace a given pattern X amount of times when a new profile is created. This pattern is referred to as a "Challenge," and it is created by referencing the Gestures API seen in **Error! Not a valid bookmark self-reference.**. As the user completes the challenges, the Gestures API will normalize the user responses and create a Profile that is associated with that challenge. This Profile will then contain the challenge along with its list of normalized responses which can authenticated against. The number of responses varies depending on what the user has set for their Strength setting. The higher the strength, the more responses required and the more precise the authentication will be. Once the user has completed all the required challenges and the Profile has been generated, the API will be able to create a User-Device Pair, which means that the authentication system should now be able to recognize the user whenever they trace the patterns.

Once the User-Device Pair has been created and the user is now trying to authenticate by tracing the challenge, the application will use its library of Python Scripts shown in **Error! Not a valid bookmark self-reference.**. to run some statistical analyses on the generated response. Over the course of the initial traces during profile creation, the application developed an average user pressure trace for authentication. There are scripts in the Python Scripts Library that will take this average trace and find a line that is 2 deviations above this average and 2 deviations below this average, thus creating a zone/distribution of acceptance. When the user tries to trace the pattern later, the trace they generate will be evaluated by a python script at 32, 64 or 128 points along the trace depending on the settings and the length of the trace. From these points, a certain percentage need to fall within the zone of acceptance for the trace to pass and for the user to be authenticated. This is illustrated in the Figure 3 and Figure 2 from Dr. Akhilesh Tyagi's team's paper [2] Figure 3 shows a trace that passes as every point fell within the zone of acceptance whereas Figure 2 shows a trace that failed as 22 out of the 32 points fell out of the zone and failed.

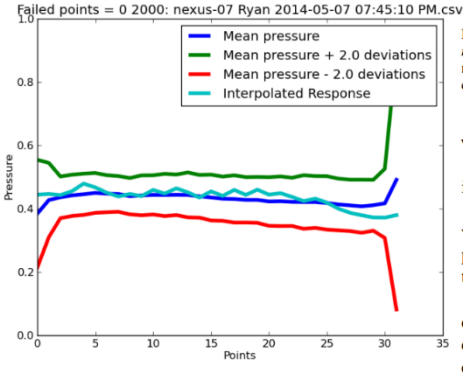

Figure 3 Statistical Concentration/Correction for a Profiled User

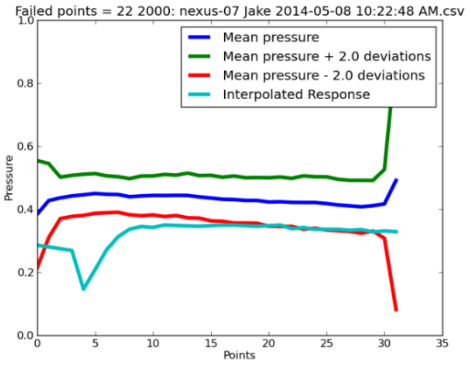

Figure 2 Statistical Concentration/Correction for a User other than Profiled User

Cryptographic Block

One major component of our application that has yet to be developed is the Cryptographic Block, which will be responsible for the encryption and decryption of our data. We plan to use Android's own Cryptography API to encrypt and de-crypt the device's file system. As shown in the block diagram, once the user trace is authenticated through the PUF Library, the application will enter the cryptographic block and de-crypt the file system. If the user is not authenticated, then the file system should remain encrypted. While we understand the high-level functionality we want, there is still more research that needs to be done in this area if we are to reach our end goal.

Currently, the goal of the solution is to integrate this application into the Android OS, so we could have application level encryption. This may look like a series of patterns opening that you must trace and pass when you open a certain application. If we can get this functionality in place, we will begin to move the encryption further back into the boot up process. Our end goal is to be encrypting and requiring user authentication at the kernel level before the OS is booted, which is like BitLocker on Windows computers. However, the feasibility of this solution is currently in question, and research is being done to see how if we could implement this feature at all. We have not explored any alternatives to this solution because this project is a research project whose purpose is to examine the feasibility and security of using a PUF in Android phones to emulate the encryption behavior of TPM chips found in laptop computers.

2.2 Design Analysis

The proposed solution has various strengths that will motivate the project's success, although there are certain drawbacks and uncertainties of feasibility.

The PUF library is the main factor of the design that makes it optimal. The research accompanied with the 4-year-old project solidifies trust in the library's stability and utility. More so than the library itself, the PUF concept in general is what makes the solution dependable. Utilizing an inherent property of circuitry and manufacturing is a practical way to circumvent the need for specialized hardware. Another major benefit is it eliminates the need to store a key, because storing a key leaves it vulnerable to malicious actors. Herder, Yu, KouShanfar, and Devadas [3] support this use of PUFs in saying they "are a promising innovative primitive that are used for authentication and secret key storage."

This is extremely accommodating to the user as tracing a pattern is already a widely accepted type of authentication on phones. It requires a slight amount of more work than the traditional Android lock screen, as setup requires the user to draw the trace multiple times. However, this takes no longer than a minute and is only done when creating a new profile. More importantly, it will also be far more secure than the old method and will not require the user to memorize the pattern they must draw. Although this will require the user to perform traces in a similar manner to be authenticated, this is something that people tend to naturally do anyway. A drawback with this approach is a user being unable to perform authentication in situations where they cannot use their usual finger. Fingerprint authentication has a similar issue, except most fingerprint scanners allow you to use multiple fingers.

However, it is still undetermined if this solution is feasible; it may not be feasible to encrypt at the kernel level. There may be a reason why Android has not shipped products with this level of encryption, such as applications sitting on top of the operating system may not be able to access what they need to. Similarly, very few resources on the subject exist that we can utilize. Although it helps to have a subset of the solution (the PUF library) already functioning to some degree, it could also be a hindrance to update and fix it, as drastic changes may need to be done to it. Although the PUF concept is established, it does not address issues like drastic changes in climate, which would affect authentication.

3 Testing and Implementation

3.1 Process Details

The teams process is simple and is explained in Figure 4 below. It consists of Consultation, Research, Design, Development, Testing and finally a solution. With guidance on when to repeat any of the listed steps.

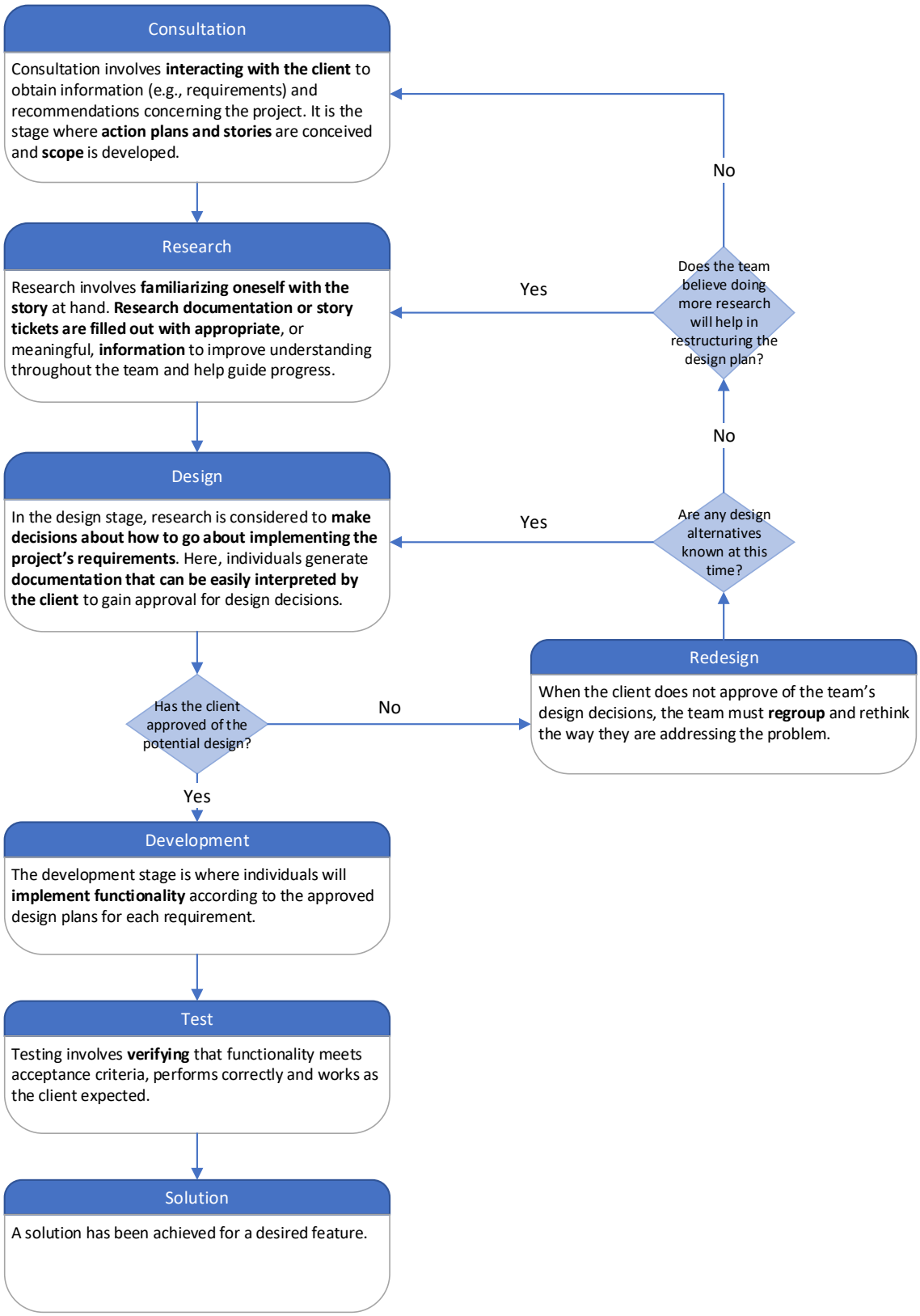

Figure 4 Process Flow Diagram

3.2 Interface Specifications

Our PUF library is not going to have an interface. It will be a JAR file that will be used within our Android application interface. The application will have different options for creating, authenticating, and deleting profiles. The application should start with the system booting and should always run in the background. It should have a clear logo and should optimize the user experience as much as possible by not using any contrasting colors, having accessibility labels, having help options for anything in the application and adjusting it as we gather more information about our users.

3.3 Hardware/Software

Most of our development so far has been Android application development, so we have been doing most of our development and testing with Android Studio, which allows us to actually deploy our application onto a device in its state at the time and also allows us to debug our application while using it which are two absolutely critical features when it comes to testing the application. When we eventually begin exploring kernel level encryption, we will begin working at the kernel level and this may require us to use additional software tools but that has yet to be determined as of this stage. In our preliminary research, we have found tools like dm-crypt which are used for encryption with Linux kernels. Since Android is built upon a Linux kernel, this may be a possible tool we could use but we still need to do further research into the feasibility of this. So as it currently stands, we plan to use Android's own Cryptography API for any encryption we will be doing (that is higher than the kernel level). We have been researching other potential options, but none seem to be quite as robust and well-integrated with the android operating system as the Cryptography API.

As far as hardware is concerned, all the initial testing that was done with the application before we took up the project was done on Nexus 7 tablets. Currently, we are specifically testing the PUF and the Android trace applications ability to identify a specific user, so we want to minimize hardware variations to really focus on the accuracy of the software. As a result, we will currently continue to test with Nexus 7 tablets however we do plan to increase our range of test hardware later in the project.

3.4 Functional Testing

Every new Java class created should have an accompanying Junit Test class that shows correct and incorrect behavior as well as demonstrates working results. This testing class should coincide with the Java class in question in file hierarchy within the testing folder for ease of

access and identification. Unit tests should be created as seen necessary to keep testing requirements unrestrictive to development time, however each new method should see some amount of coverage. These Junit test cases will be conducted as necessary for development of components, as well as during acceptance testing to prove correct operation. Dependency injection will be utilized using the Mockito library to stub external components to isolate the tested component. The results are easily interpreted by the pass or fail output of the Junit tests, through use of test case assertions using the Junit framework.

Integration testing should begin once component dependencies start to show in new features, and integration tests should be designed for expected results with the intent of real data being returned from the depending component. Integration testing should be designed with only between required components in a new functionality, with the remaining existing components mocked or stubbed using dependency injection to keep the test functional requirements isolated and reliably tested like unit testing. Integration Testing will be conducted.

System testing of the proposed solution and its requirements will be initiated as multiple components of the solution approach full functionality. System tests will primarily be created by the test engineer of the team, however other members of the team most familiar with specific components are invited to aid in creating parts of end-to-end testing as well. System tests are end-to-end and will allow us to demonstrate that all functional requirements are enough in a live environment on the Nexus 7 tablet.

Acceptance testing will be down in two stages:

1. Tickets are reviewed at the closing of a ticket where Unit and any Integration tests are validated.
2. Components are verified at the end of their completion to ensure all desired functional requirements are met and pass all system tests.

Test Cases:

Functional Requirement 1: Application cannot be closed when application is encrypting

Test Case:

Step 1: user A has an existing authentication profile, or otherwise creates one.

Step 2: user A shuts down and starts up the device.

Step 3: during startup, user A attempts to close the application at boot.

Acceptance Criteria: User A should be prompted with a message notifying that the device cannot be shut down by normal means until the encryption steps at boot for the application complete

Functional Requirement 2: Users should be able to create multiple profiles

Test Case:

Step 1: user A creates a user profile with the application.

Step 2: user B creates a separate user profile with the application.

Step 3: both users attempt to access user data one at a time, each tracing the authentication pattern when prompted.

Acceptance Criteria: Both User A and User B should be able to create their own profile successfully and be able to be properly authenticated through their respective profiles and be given access to their user data.

At least 95% acceptance is expected while less than 1% of the time providing access to the user via the another's authentication profile.

3.5 Non-functional Testing

There are several non-functional requirements that must be met in acceptance testing to verify the product.

- **Performance:** Testing will require the actual use of the Nexus 7 tablet for real world scenarios. Authentication systems are heavily reliant on the speed of the device alongside the optimization of the authentication process.

  Test Case: Full authentication should take no longer than 4 seconds to complete

  Step 1: user A has an authentication profile, or otherwise creates one

  Step 2: user A attempts to access user data, and traces the shape prompt

  Step 3: feedback is received by user A, revealing the result of the request

Acceptance Criteria: Authentication is successful within 5 seconds. In the case of this requirement not being passed, further optimization of the code may be necessary.

- **Scalability:** being able to extend the application's features past their basic implementation is important for growing its usefulness

  Test Case: More than 2 authentication profiles are can be stored

  Step 1: user A creates an authentication profile

  Step 2: user B creates an authentication profile

  Step 3: both user attempt to access user data, and trace the shape prompt

  Acceptance Criteria: Both user A and user B are given access to user data as their individual authentication profiles were both saved. Failure results in analysis of the authentication profile storage system.

- **Maintainability:** Ensuring our application's state can be handled and controlled without time-consuming effort is important for consistent development.

  Test Case: Pushing an update to the repository to update the application

  Step 1: Changes are committed to the application

  Step 2: Developer Operations are set to automatically build an updated version

  Step 3: The updated version is automatically uploaded to update the application

  Acceptance Criteria: Project repository can update the version of the application automatically and without human intervention for completion. Failure results in modification of the developer operations deployment settings.

- **Security:** protecting and managing data access is the applications most vital attribute.

  Test Case: Accessing data while unauthorized

Step 1: user A attempts to access user data without an authentication profile

Step 2: user A traces the shape prompt

Acceptance Criteria: User A is not given access to the user data and is given notification of their denial. Adverse results result in analysis of the authentication and profiling systems.

- **Data Integrity:** Successfully encrypting and decrypting files with comparisons of the data before and after this process to ensure that the integrity of the data is maintained for the user is dire to the operation of the application.

  Test Case: Data is correctly encrypted and decrypted without corruption

  Step 1: A file is included in user data, and is encrypted

  Step 2: The same file is accessed, and decrypted for viewing

  Acceptance Criteria: The file should not be subject to corruption during either encryption or decryption. Failure to keep file integrity requires analysis of the encryption and decryption processes.

3.6 Modeling and Simulation

To assist in testing the application, our team uses Android Studio to model the layouts of activities throughout the program. Doing so helps ensure the accuracy of the client's desired application appearance. Additionally, Android Studio provides access to an emulator which simulates a user utilizing the application and allows developers to easily test functional requirements. By using this form of simulation, program behavior and a user's experience can be quickly evaluated. Emulation provides an efficient way to conduct tests; however, using an emulator is not always realistic. An emulator does not appropriately represent the functionality of a PUF. To combat the issue of realism, our team also uses physical Android devices. Using this type of device allows for the collection of empirical data, ultimately helping to verify and validate that the implemented functionality works appropriately.

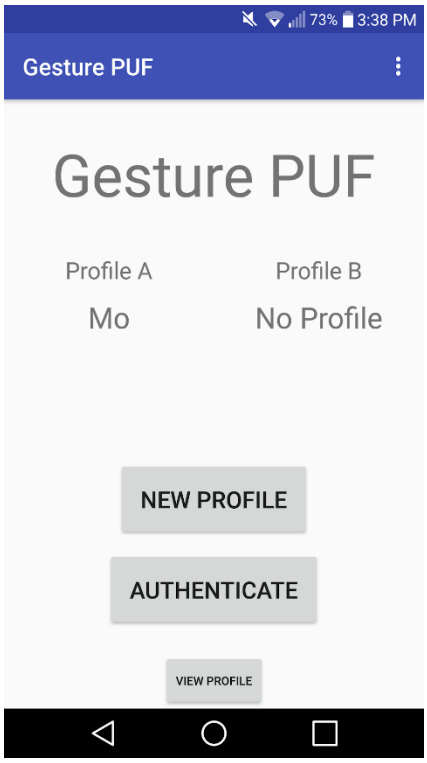

Figure 7 Model of layout

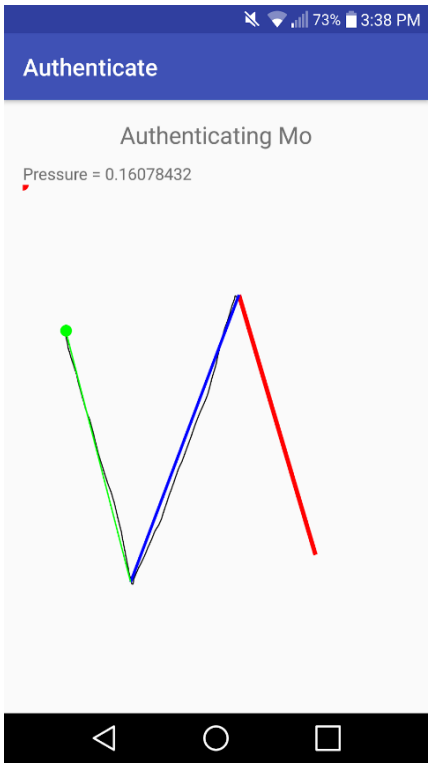

Figure 6 Simulation of trace

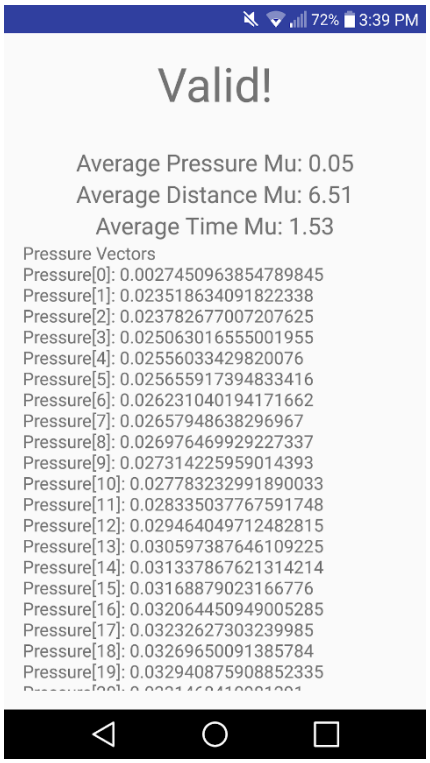

Figure 5 Simulation results

3.7 Implementation Issues and Challenges

There are several implementation issues and challenges related to development of this project. By design, once authentication profiles for each user are created, they are tied to the device they are created on. Therefore, the testing environment is not very mobile, which makes this very difficult as each set of profiles must be tested on an exact device. In this same way, we also cannot develop the application for any other devices in the project's current state as authentication is designed for use with the nexus 7's hardware.

However, although profiles need to be tested on the device that they are tied to, this is not too difficult as these are mobile devices. This would be a larger issue in an environment with a very large team or if team members were not physically located near each other. To get around this issue, we simply share the test devices among the team. During actual use cases, this limiting factor is something the user is encouraged to account for when protecting their data.

There are some challenges with inheriting a 4-year-old library. One observation made was a heavy use of hard-coded values, which makes updates and patches hard to perform. Naturally, fixing issues is much more manageable than creating a brand-new solution. The lifetime of the

library has also resulted in multiple implementations of normalization, key generation, and authentication. This could make it easy to switch between implementations based on needs at a given time, or it could make it hard to create a functioning one, as piecing together different parts could be a challenge. To accommodate with this issue, some areas of the library will need to be updated and streamlined for future development.

Another issue is the level of encryption we can implement within the android SDK. It is unclear yet whether we can implement the application as the operating system is booting for an ideally lower level of protection. It's also unclear if implementing full disk encryption is a feasible solution for this application, or whether it would create an unfriendly user experience requiring constant user authentication from the user by the operating system. Android used to support full disk encryption and then switched to file-based encryption, alluding to issues with full disk encryption, or at least removed support for it. To solve this uncertainty, more research and foresight for system design must be used to work around these limitations.

However, implementing full-disk encryption on Android is more a question of feasibility not possibility, as Android used to support it. Android does not specifically say why they switched to file-based encryption but explain that development is less tedious as applications can be loaded without a password. If full disk encryption does require authentication for individual applications after the phone is unlocked, a supplementary solution could be providing a PIN after being authenticated. This way, PUF is still the primary method for authentication, but a very quick form of credentials is passed instead of performing the cumbersome trace.

4 Closing

4.1 Conclusion

Overall, the purpose of this project is to create a more secure way to protect data on a user's phone. The project will be implemented by using the design of our client in the form of a PUF. Using the private key dynamically generated by the PUF to encrypt the user's data. Data will be encrypted using android's already existing encryption API. At the very end of next semester, given all assumptions are upheld, a Nexus 7 should be able to be fully encrypted and decrypted using the PUF.

References

[1] "Subscriber share held by smartphone operating systems in the United States from 2012 to 2018," Statista, 2018. [Online]. Available: https://www.statista.com/statistics/266572/market-share-held-by-smartphone-platforms-in-the-united-states/. [Accessed 1 12 2018].

[2] A. T. Ryan A. Scheel, "Characterizing Composite User-Device Touchscreen Physical Unclonable Functions (PUFs) for Mobile Device Authentication," in *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*, Denver, Colorado, USA, 2015.

[3] C. Herder, M.-D. Yu, F. Koushanfar and S. Devadas, "Physical Unclonable Functions and Applications: A Tutorial," *Proceedings of the IEEE,* vol. 102, no. 8, pp. 1126-1141, 2014.