Biometric Bit locker

Design Document

Team: 23

Client/Advisor: Akhilesh Tyagi

Yousef Al-Absi, Cole Alward, Morgan Anderson, Ammar Khan, Justin Kuhn, Larisa Thys

sdmay19-23@iastate.edu

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

Revised: 04/24/2019

Table of Contents

Figures

1 Introduction

1.1 Problem Statement

Asymmetric encryption is an encryption that allows individuals to hide their data and ensure it is secure. Typically, files would be decoded using a public key and then encoded using a private key. The private key is stored in a Trusted Platform Module (TPM), which is a cryptographic module that enhances computer security and privacy. TPM chips are usually discrete chips soldered into a computer's motherboard, allowing for separation 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 Physical Unclonable Function (PUF). In doing so, the dynamically generated private key will not be stored anywhere on the device and will be able to authenticate against the public key. The key can only be generated at runtime, solving the issues of not having a TPM and storing the key.

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.

According to Statista [1], 54.1% of people in the United States are using an Android device. Therefore, we would want to reach the Android market with our application since most people today have 2 data on their phone worth encrypting. For example, any employees within a company that keep sensitive data on their phone may wish to keep their data encrypted in case their device is stolen or compromised.

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 received from our client. Our hope is to fix the issues we have found in the library by the beginning of next semester so we can use the working PUF to develop our application.

- Android continues to support full-disk encryption.

- Android currently supports full-disk encryption. However, due to the legality associated with encrypting the full-disk, Android is considering suspending support for 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, the application will not reach its full potential and will have to reside at the application level.

Limitations:

- Nexus 7 hardware.

  The Nexus 7 is the hardware the school provides and is the device the PUF was originally designed on. The team decided it would be best to continue implementation on this hardware. Unfortunately, this will provide some limitations. The Nexus 7 currently handles API 23, which is a much lower API 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 imperative to use it within the project.

1.5 Project Goals and Deliverables

Goals:

- To continue development on the provided 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 then authenticate 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.
    - The library should have at least 70% test coverage.
- A PUF-based Android application
    - The application should act as a lock screen whenever the phone is closed. It will authenticate users by asking them to draw a shape, and the application should only authenticate and decrypt information for the correct user. The application will also automatically start when the phone boots.
    - Authentication should happen within 5 seconds.
    - The application should work on phones with 1GB of RAM.
    - Application size should not exceed 80 MBs.

2 Design Specifications

As the PUF library was provided for us, we are not designing anything new for it. Rather, we will be fixing and maintaining the library to ensure it meets the requirements listed below and incorporating the library into an Android application of our making.

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 it is locking phone.
- PUF should encrypt and decrypt user profiles.

Non-Functional Requirements:

- Performance: Response time for authentication should be less than 5 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 when provided the correct key.
- PUF should have an accuracy of at least 80%.

2.1 Proposed Design

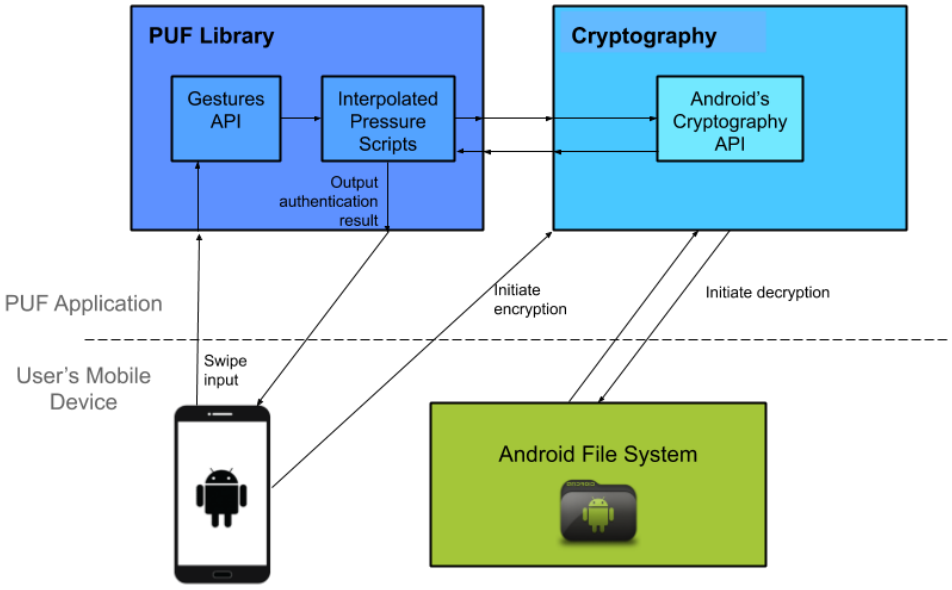

FIGURE 1 INITIAL DESIGN OVERVIEW

For our project, some of the groundwork for the solution has already been completed. The current solution is to develop an Android application using data generated from pressure readings returned by the screen when the user traces a generated shape similar to 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 TPM chip which can be found on most laptop computers. This enables full-disk encryption on laptop computers, where 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 the TPM 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, thus eliminating the need to save it. The key generation will occur when the user traces a pattern generated by the application. When provided with the pattern, the application requires users to trace the shape a certain amount of times depending on the selected user and selected strength (i.e., the higher the strength, the higher the number of

traces). Doing so will 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 a given device is unique, 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. Considering this, the implementation will lead to both user authentication and device authentication, thus leading to maximal security with the given hardware.

PUF Library Block

The pattern tracing and user authentication features are all encapsulated within the "PUF Library" subblock in **Figure 1**. These features have all been implemented; however, they 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 **Figure 1**. As the user completes the challenges, the Gestures API will normalize the user responses and create a profile associated with that challenge. This profile will then contain the challenge along with its list of normalized responses, which can be 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 **Figure 1**, 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 and below this average, thus creating a zone, or 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, which is dependent 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 user to be authenticated. This is illustrated in Figures 4 and 6 from Dr. Akhilesh Tyagi's team's paper, which are shown in **Figure 3** and **Figure 2** below. [2] **Figure 3 Statistical Concentration/Correction for a Profiled UserFigure 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.

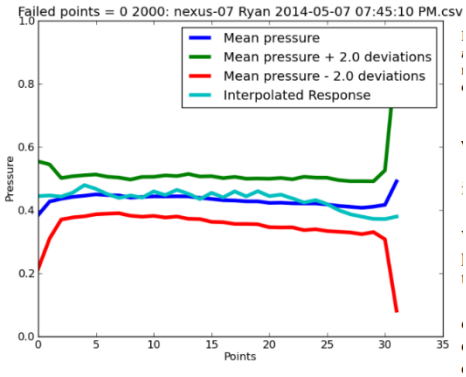

FIGURE 3 STATISTICAL CONCENTRATION/CORRECTION FOR A PROFILED USER

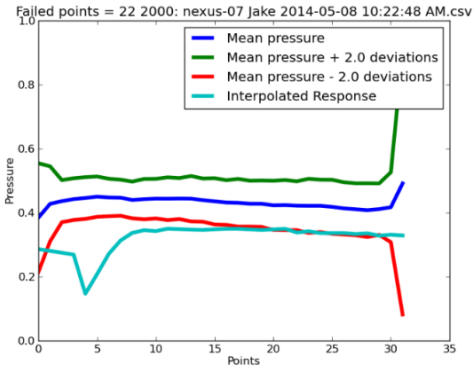

FIGURE 2 STATISTICAL CONCENTRATION/CORRECTION FOR A USER OTHER THAN PROFILED USER

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 decrypt 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 the gestures application into the Android OS so application level encryption is available. This may look like a series of patterns that you 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 whether we can 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 pressure-based 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 motivate the project's success; however, certain drawbacks and uncertainties of feasibility exist.

The PUF library is the main, and most optimal, component of the design. The research accompanied with the four-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 the PUF library provides is that it eliminates the need to store a key. Storing a key leaves data 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."

The PUF's functionality is extremely accommodating to the user, for tracing a pattern is already a widely accepted type of authentication on phones. Utilizing a PUF requires more work from the user than simply using the traditional Android lock screen, as setup requires the user to draw the trace multiple times. However, the tracing process takes no longer than a minute and is only done when creating a new profile. More importantly, these traces will be used to make data more secure than the old method and will not require the user to memorize the pattern they must draw. The only requirement is that the user must perform traces in a similar manner to be authenticated, which is an action people tend to naturally do anyway. A drawback with the pressure-trace approach is that a user may be 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.

Unfortunately, the feasibility of our solution remains unknown; 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, one scenario being that applications sitting on top of the operating system may not be able to access what they need to. Additionally, very few resources on the subject exist that we can utilize. While it helps to have a subset of the solution (i.e., 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 team's workflow process, represented in **Error! Reference source not found.** below, is simple, yet effective, in learning and implementing new ideas. In an ideal situation, the process consists of five stages leading up to the solution. However, to account for potential disagreements, the workflow process also provides guidance on how to handle conflicting situations so progress can continue to be made.

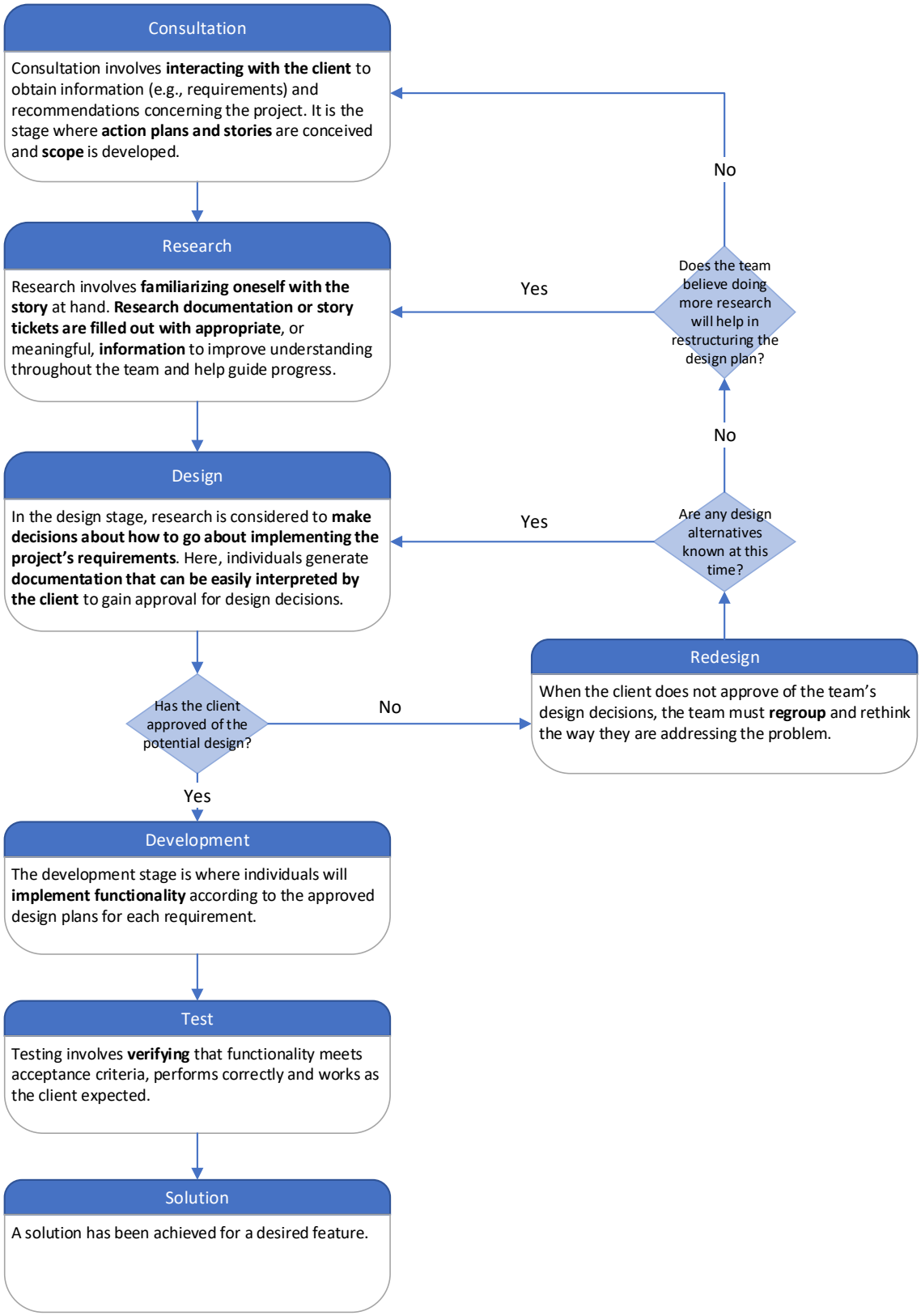

FIGURE 4 PROCESS FLOW DIAGRAM

3.2 Interface Specifications

The PUF library will not have an interface. Rather, the library 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. Additionally, the application should have a clear logo and optimize the user experience as much as possible by avoiding any contrasting colors, incorporating accessibility labels and providing "help" options for anything in the application. As we gather more information about our users, the program will be adjusted.

3.3 Hardware/Software

Software

Presently, much of the project's development has centered around the advancement of the Android application. To do so, our team has been utilizing Android Studio to expand upon the client-provided PUF library and gestures application and deploy the said application onto a device for modeling and simulation. Additional details on modeling and simulation are provided in **Section 3.6** Modeling and Simulation.

When it is time to focus on incorporating kernel level encryption in our project, we may be required to use additional software tools to implement the desired functionality. One tool we discovered in our preliminary research of kernel level encryption was "dm-crypt," which is a disk encryption subsystem used in Linux kernels. Since Android devices are built upon the Linux operating system, "dm-crypt" is a potential candidate for assisting in implementation of cryptography at boot. Although dm-crypt meets our project's needs, our team needs to do further research on the feasibility of the subsystem working with our current constraints.

While part of our team investigates kernel level encryption, the project will, for the time being, utilize Android's Cryptography API at the application level for any encryption and decryption we will be doing on user data. Based on our research, the API appears to be the best solution thus far, providing a robust solution that is also well-integrated with the Android operating system.

Hardware

Prior to our team's contribution to this project, all initial testing for the application was performed on Nexus 7 tablets. Currently, we are focusing our efforts on testing the PUF library and Android gesture application's ability to authenticate a specific user, so we want to minimize hardware variations to evaluate the accuracy of the provided software. We will continue to test

with Nexus 7 tablets; however, we plan to increase our range of hardware in the future to evaluate our program's effectiveness on other devices.

3.4 Functional Testing

Each newly created Java class should have an accompanying JUnit test class that shows correct and incorrect behavior and demonstrates working results. The folder structure within the test folder should mirror the folder structure in the source folder for ease of access and identification. Unit tests should be created whenever 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 needed for development of components as well as during acceptance testing to prove correct operation. Dependency injection will be utilized using the Mockito library to simulate external components to isolate the tested component. The results will be easily interpreted by the pass or fail output of the JUnit tests by using test case assertions via 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.

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

Acceptance testing will be done in two stages:

1. Tickets will be reviewed at the closing of a ticket where unit and any integration tests are validated.
2. Component functionality will be verified upon completion to ensure all desired functional requirements are met and all system tests pass.

Test Cases:

Functional Requirement 1: Application cannot be closed during encryption.

Test Case:

Step 1: User A has an existing authentication profile; otherwise, a profile is created.

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 are 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, be properly authenticated through their respective profiles and be given access to their user data.

A user should be able to access his or her data 95% of the time, and the user should not be able to login to other users' accounts 99% of the time.

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 use of the Nexus 7 tablet for real world scenarios. Authentication systems are heavily reliant on the speed of the device and the optimization of the authentication process.

  Test Case: Full authentication should take no longer than 5 seconds to complete

  Step 1: User A has an authentication profile; otherwise, a profile is created.

  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. If this requirement is not fulfilled, further optimization of the code may be necessary.

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

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

  Step 1: User A creates an authentication profile.

  Step 2: User B creates an authentication profile.

  Step 3: Both users attempt to access their user data by tracing the shape prompt.

  Acceptance Criteria: Both User A and User B are given access to their respective user data, for 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 application's 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 outcomes 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 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.

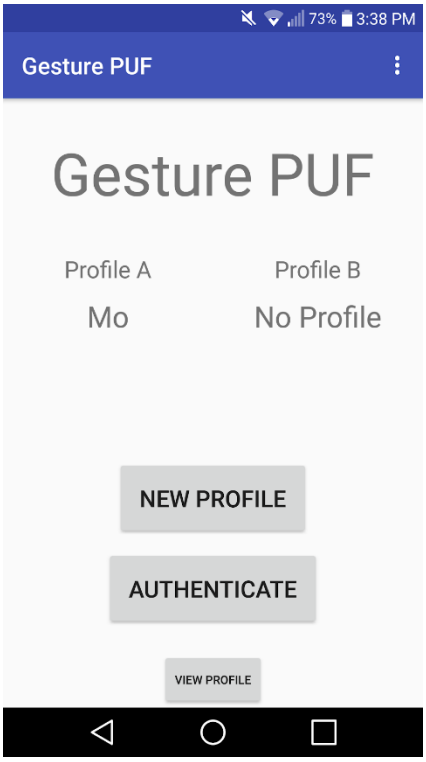

FIGURE 7 MODEL OF LAYOUT

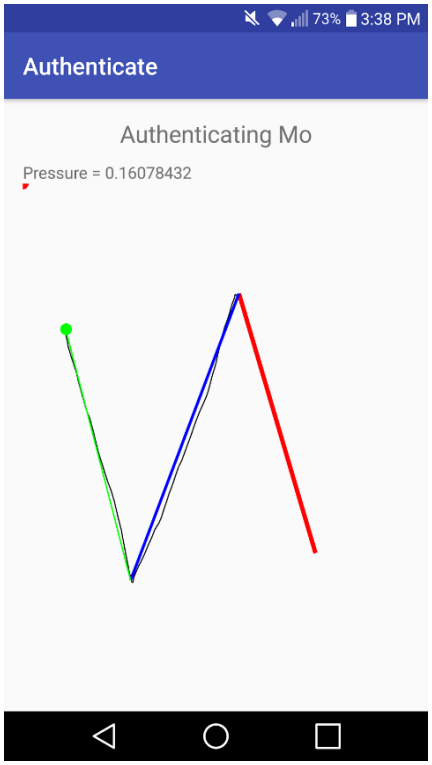

FIGURE 6 SIMULATION OF TRACE

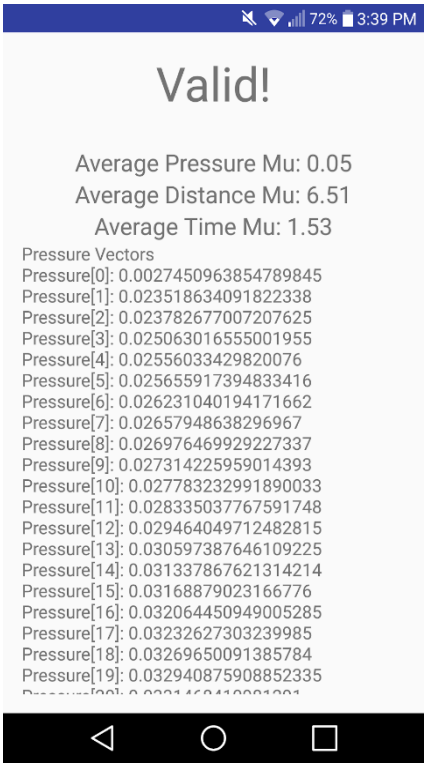

FIGURE 5 SIMULATION RESULTS

3.7 Implementation Issues and Challenges

There are several implementation issues and challenges related to the 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 versatile, which makes evaluation very difficult as each set of profiles must be tested on an exact device. In the 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.

Although profiles need to be tested on the device that they are tied to, evaluation is not too difficult, for we are working with mobile devices. The situation would be a bigger issue in a scenario involving a very large team or where team members were not physically located near each other. To get around this issue of limited hardware resources, we simply share the test devices among the team. When considering actual use cases, this limiting factor is something the user is encouraged to account for when protecting their data.

In addition to being tied to a device, there are some challenges that accompany the inheritance of a four-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. Having these various solutions could make it easy to switch between implementations based on needs at a given time, or they could make it difficult to create a functioning program, as piecing together different parts could be a challenge. To mitigate this issue, some areas of the library will need to be updated and streamlined for future development.

Another problem we have encountered is the level of encryption we can implement within the Android SDK. It remains unclear whether we can implement the application as the operating system is booting for an ideally lower level of protection. It is also unclear if implementing full-disk encryption is a feasible solution for this application or if 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 but 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.

Considering the problem above, 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 they 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 provided by our client in the form of a PUF. Using a private key dynamically generated by the PUF, a user's data will be encrypted using Android's pre-existing Cryptography API. At the 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.