Mobile, Biometric Bitlocker

Final Report

29 April 2019

Team Number: 23

Team Email: sdmay19-23@iastate.edu

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

Client/Advisor: Akhilesh Tyagi

Developers: Yousef Al-Absi

Cole Alward

Morgan Anderson

Ammar Khan

Justin Kuhn

Larisa Thys

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 Solution

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 data on their phone worth encrypting. For example, any employees within a company that keep sensitive data on their mobile device may wish to keep their data encrypted in case their device is stolen or compromised.

1.4 Project Goals and Deliverables

The goals of this project included the following:

- Continued development on the provided open source PUF library.
- To make the PUF function as a lock screen by requesting a user to draw shapes to unlock the phone and then authenticate properly.

This project delivers the following products:

- **A well-tested PUF Java "Gestures" library**: An open source library released with an appropriate architecture for the software and a valid testing framework as well as reworked methods. This library has at least 70% test coverage.
- **A PUF-based Android application**: An application that authenticates users by asking them to draw a shape, validates whether the user is correct based on collected pressure data and decrypts information depending on the validation results.

2 Requirements Specifications

To deem this project as a success, the Android application must meet the following functional and non-functional requirements to fulfill the client's desired use cases:

Functional Requirements:

- Application should be able to create multiple profiles.
- Application should be able to authenticate users using PUF.
- Application continues operation when phone is locked.
- Application should encrypt and decrypt files based on a user's profile.

Non-Functional Requirements:

- Performance: Response time for authentication should be less than 5 seconds. PUF should have an accuracy of at least 80%
- 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.

3 System Design and Development

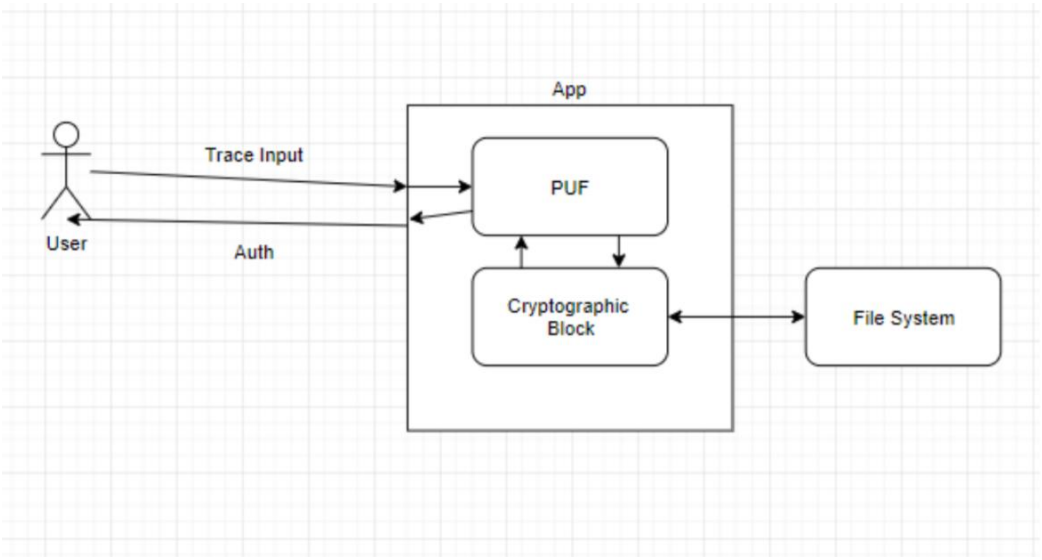


FIGURE 1 INITIAL CONCEPTUAL DESIGN

3.1 Proposed Design

For our project, some of the groundwork for the solution had already been completed. Our solution involved the development of an Android application that uses 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 is generated by a PUF, which was implemented by a previous team.

As stated earlier, mobile devices lack the TPM chip which can be found on most laptop computers. The chip 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 no 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 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.

3.2 System Constraints

The main system constraints we experienced with this project were related to the Android operating system (OS) and the actual hardware used for testing the pattern-tracing "Gestures" application. Regarding the former, we experienced issues with the Android OS's constraints around encryption. As of Android 7.0, Google has decided to move away from full-disk encryption (FDE) in favor of file-based encryption (FBE). This caused issues for our design because we wanted to emulate a TPM, which would fully encrypt the device. To get around this, we tried to implement a Kiosk mode type application that would essentially require the user to authenticate through PUF and encrypt the desired files at the lock screen level. Unfortunately, Android has yet to develop the Kiosk mode to the point where an implementation of this nature would be possible.

In terms of the pattern tracing library, we were severely limited in our choices of hardware because the pressure readings used by the application for user and trace authentication were screen dependent. In other words, the pressure readings from a Nexus 7 tablet could vary greatly from the pressure readings of a Samsung smart phone due to different screen technologies used in each device. We experienced this issue when testing on a variety of devices such as the Nexus 7, Samsung Galaxy S10 and OnePlus 5T. For the purposes of our project, we chose to move forward using the Nexus 7 as our main device, for it was the device used by the previous team working on this project. Continuing with the Nexus 7 as our device of choice as opposed to switching to another device provided less variability and hurdles for us moving forward.

3.3 Design Trade-offs

To make our project successful, we were forced into trade-offs concerning our implementation of encryption and where the PUF application would be used in the Android OS lifecycle.

As previously mentioned, the application was targeting the Android OS; however, as of version 7.0, the OS has moved away from FDE to FBE. From Android version 4.4 to 6.0, the OS used to ship with a kernel device mapper called "dm-crypt," which was designed to encrypt at the sector level, resulting in FDE. This tool and functionality as a whole, however, were both removed, so we had to modify our approach. Since the main purpose of this project was to expand upon research and act as a proof of concept, we believed it would be appropriate to continue implementing encryption on a file-based level to illustrate the fact that the provided

PUF technology could be feasibly utilized to generate keys used in encryption and decryption of data.

Regarding where the PUF application would operate, we had to reassess our goals several times. Initially, our objective was to execute the application on boot and appear before the OS was loaded, much like Bitlocker for Windows. However, further research and exploration of certain tools showed us that this would not be a feasible goal. We then shifted our goal to move the application into what Android calls “Kiosk mode,” which is a mode phones are often found in at stores and when first powered on. When a phone is first activated, the user cannot access the OS until they have navigated through all the setup screens. Our hope was to require the user to complete a trace and authenticate within Kiosk mode before they could access the OS, but further research and attempted implementations demonstrated that this would not be possible given that Kiosk mode has extremely limited capabilities. The technology is currently not in place for us to be able to develop what we wanted to achieve, so we, again, shifted our goal to keep the PUF library at the application level. This could be changed in the future if Google decides to expand upon Kiosk mode or bring back the previous implementation of FDE with dm-crypt as an option.

3.4 Architectural, Design Block Diagram

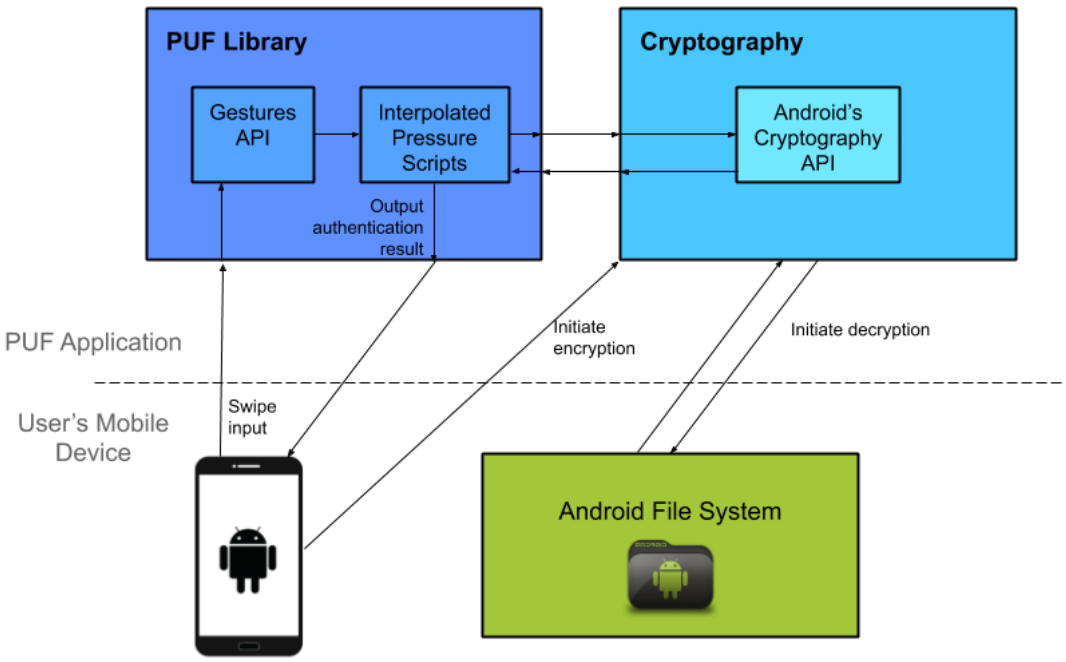


FIGURE 2 DESIGN OVERVIEW

3.4.1 PUF Library Block

The pattern tracing and user authentication features were all encapsulated within the "PUF Library" subblock in Figure 2**.** These features had all been previously implemented; however, they needed to be updated, reworked and thoroughly tested before we deemed 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 2**.** As the user completes the challenges, the Gestures API normalizes the user responses and creates a profile associated with that challenge. This profile contains 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 completes all the required challenges and the profile has been generated, the API is then able to create a User-Device Pair (U-DP), which means that the authentication system is able to recognize the user whenever they trace the patterns.

Once the U-DP is created and the user tries to authenticate by tracing the challenge, the application will use its library of interpolation pressure scripts, shown in Figure 2, to run some statistical analyses on the generated response. Over the course of the initial traces during profile creation, the application develops an average user pressure trace for authentication. There are scripts in the interpolation pressure scripts library that 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 needs 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 [2], which are shown in Figure 3 and Figure 4 below. Figure 3 shows a trace that passes as every point fell within the zone of acceptance whereas Figure 4 shows a trace that failed as 22 out of the 32 points fell out of the zone and failed.

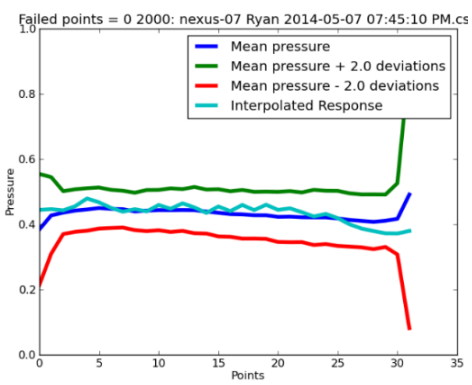


FIGURE 3 STATISTICAL CONCENTRATION/CORRECTION FOR A PROFILED USER

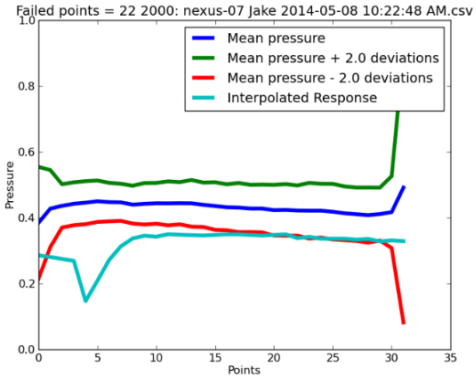


FIGURE 4 STATISTICAL CONCENTRATION/CORRECTION FOR A USER OTHER THAN PROFILED USER

3.4.2 Cryptographic Block

One major component of our application our team developed was the "Cryptographic Block," which is responsible for the encryption and decryption of our data. We used Android's own Cryptography API to encrypt and decrypt the device's file system. As shown in the block diagram, once the user trace is authenticated through the PUF Library, the application enters the Cryptographic Block and decrypts the file system. If the user is not authenticated, then the file system remains encrypted.

4 Implementation

4.1 PUF Library

The largest portion of the development phase of our project was updating and maintaining the PUF library. The most extensive rework within the library was modifying where the collected pressure data was being processed. Originally, the data was processed within Python scripts. Since the eventual goal of this project is to integrate our application at the kernel level, we thought the data processing should be done in Java. Therefore, to streamline the process of using these scripts in a Java dominated environment and improve our understanding of these scripts, we altered the library by making it into a JAR and dynamically translating the code with a Python converter for the Android application to use.

While translating the Python scripts to Java, we identified several flaws in how the initial methods were implemented. We changed these methods to create a better implementation that more accurately reflected our client's research. [2] Since the goal of this application is to be a proof of concept, we deemed it worth our time to perfect the methods within the library to highlight the full potential of the desired implementation.

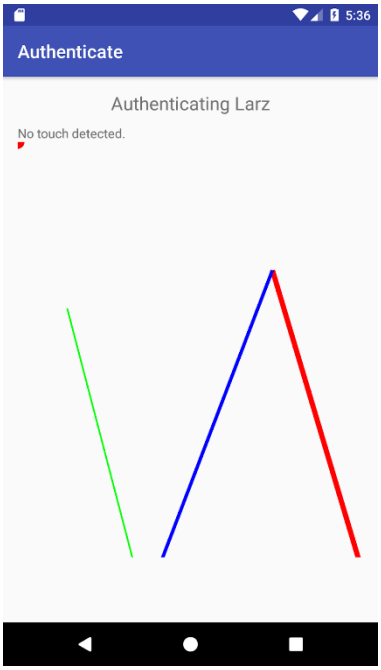


FIGURE 5 OUT-OF-BOUNDS CHALLENGE

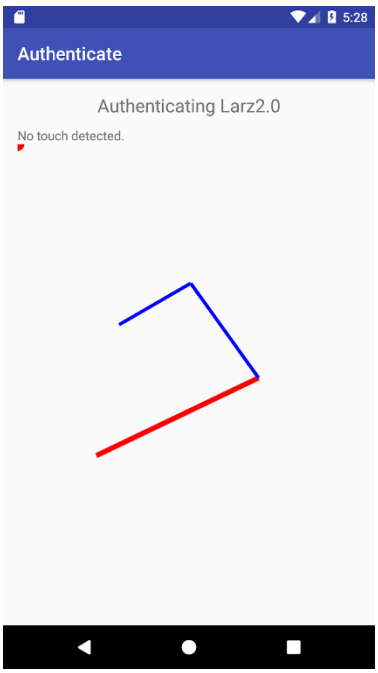


FIGURE 6 CORRECTED CHALLENGE

Another change we made to the implementation was updating the challenges to follow the intended plan. As you can see from Figure **5,** the intersection of the green and blue lines is out-of-bounds, or outside the trace area. Additionally, these challenges would occasionally cross itself. Both of these cases contradicted the research done and, thus, needed to be fixed. To do so, we implemented the challenges to consider the screen space and generate a point in each quadrant of the area, so the line would never cross itself (Figure 6).

4.2 Cryptography

The Cryptographic Block is an extension of the Android cryptography API to encrypt and decrypt the application's files. The encryption method is the Advanced Encryption Standard (AES) with cipher block chaining. We used a 128-bit key from the normalized and quantized data provided by the PUF library. A new initialization vector is created and written to a file every time data is encrypted, and the decryption process uses the saved initialization vector. The key is made at runtime and is used to decrypt the application's files if it is correct; otherwise, the application will keep the files encrypted. When the user exits the application (i.e., goes "home," "back" or locks the device), the application re-encrypts all application-level files.

4.3 Mobile Application

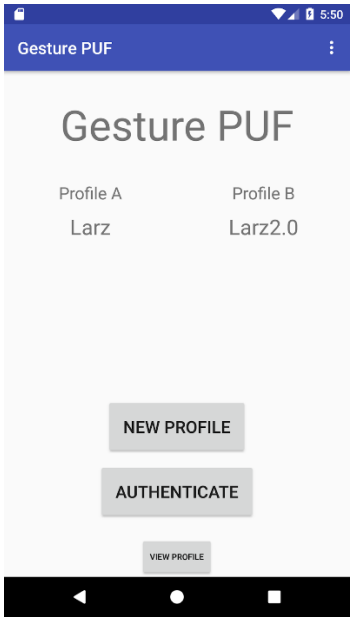


Figure 7 Users' Profiles Before Changes

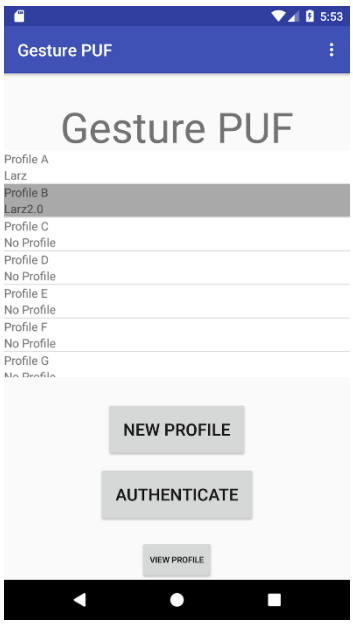


Figure 8 Users' Profiles After Changes

The mobile bitlocker application is an extension of another application we received from our client. Upon receiving the application, the program only demonstrated whether the challenge functioned properly. Our initial goal with this component was to wrap another application around the challenge activity that would allow the user to see his or her files. We did this by adding extra activities to the application, which only appear when the user is successfully authenticated by the PUF. It is a simple implementation that allows the user to see the files that are decrypted by the application.

In addition to the extended authentication and encryption activities, we added the ability to create up to ten different profiles for test purposes. A user's profile is what stores the data generated from profile creation, where a user traces a given pattern an arbitrary amount of times to distinguish him or her from a different U-DP. This was accomplished by using a list view to prevent the profiles from taking up more space than was already allocated for the profile section of our application.

4.4 Applicable Standards

4.4.1 Agile

The team followed Agile planning and development methods during this project. We worked in weekly sprints, where meetings were held semiweekly for one hour to discuss progress on assigned issues. Each week, members were assigned a new issue or carried issues from the

previous sprint to continue work. Depending on the size of the issue, multiple developers may have worked together to improve efficiency.

Meetings opened with a stand-up so all members were aware of project progress and given an opportunity to offer feedback or advice. Afterwards, the team refined the backlog, updated the status of open tickets and opened new issues for future sprints. Weekly status reports were also produced at this time to inform the client of progress and future documentation.

This process worked well for us, aiding in keeping project progress moving consistently and in maintaining regular time periods to communicate with each other.

4.4.2 Test-Driven Development

The team tried to follow the IEEE 1619-2007 standard when implementing encryption in our application.

4.4.3 Test-Driven Development

When we received the PUF library, our client assumed the software was working appropriately. Unfortunately, we discovered that a lot of additional development had been done and the library did not operate as expected. One of the primary reasons for this issue was that many of the features had not been tested at all. The library was relatively large and had many interconnected components, making it difficult to determine which components were working and which components were causing issues. This was an issue our team did not want to replicate; therefore, we decided that new features should be test-driven, and we should strive for as close to 100% test coverage as possible.

5 Testing, Validation and Evaluation

5.1 Test Plan

5.1.1 Unit Testing

Each newly created Java class had an accompanying JUnit test class that showed correct and incorrect behavior and demonstrated working results. The folder structure within the test folder mirrored the folder structure in the source folder for ease of access and identification. Unit tests were created whenever necessary to keep testing requirements unrestrictive to development time; however, each new method saw some amount of coverage. These JUnit test cases were conducted as needed for development of components as well as during acceptance testing to prove correct operation. Dependency injection was utilized using the Mockito library to simulate external components to isolate the tested component. These simulations allowed

the developer to define static function to components that were not fully implemented or needed to be overwritten for desired test input. The results were easily interpreted by the pass or fail output of the JUnit tests through the use of test case assertions via the JUnit framework.

5.1.2 Integration Testing

Integration testing began once component dependencies started to show in new features. Integration tests were designed to verify expected results with the intent of real data being returned from the depending component. Additionally, integration testing was designed to contain only required components sharing new interactions, with the remaining existing components mocked or stubbed using dependency injection to keep the functional requirements isolated and reliably tested.

5.1.3 System Testing

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

5.1.4 Acceptance Testing

Acceptance testing was done in two stages:

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

5.2 Validation and Verification

Each project requirement required a system test case for validation and needed to pass the acceptance testing criteria to be verified as sufficiently implemented. All acceptance test cases required manual system testing utilizing the hardware device to simulate user-level interactions. Depending on the nature of the component being tested upon feature completion, verification testing may have been proven with Java Virtual Machine unit tests.

All versions and branches of development must have passed the previously implemented unit, integration and system tests to be considered stable, ensuring no features were lost in version-controlled development. Core implementation features were validated and verified by multiple team members who achieved similar results to ensure consistent operation.

5.3 Evaluation

5.3.1 Functional Testing

The following functional requirement use cases were met in acceptance testing to validate and verify the product:

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 locks the phone as it is encrypting.

Acceptance Criteria: The application should continue to complete its intended operation, without error, in the background before closing.

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 80% of the time, and the user should not be able to login to other users' accounts 98% of the time.

5.3.2 Non-Functional Testing

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

- **Performance:** Testing required 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.

6 Project and Risk Management

6.1 Task Decomposition

- Yousef Al-Absi
    - Responsibilities:
        - Understood Gradle
        - Assisted with PUF issues
        - Implemented DevOps
- Cole Alward
    - Responsibilities:
        - Implemented encryption
        - Assisted in application integration
        - Organized ticket flow
- Morgan Anderson
    - Responsibilities:
        - Implemented key generation
        - Assisted in application integration
        - Aided in technical writing and schedule organization
- Ammar Khan
    - Responsibilities:
        - Interacted with client
        - Assisted in rewriting interpolated pressure scripts
        - Aided others with PUF issues
- Justin Kuhn
    - Responsibilities:
        - Developed test plan
        - Conducted test integration
        - Assisted in rewriting interpolated pressure scripts
- Larisa Thys
    - Responsibilities:
        - Led semiweekly meetings
        - Assisted in reworking authentication
        - Aided others with PUF issues

6.2 Project Schedule

The schedule for this project was based on the rolling wave planning technique, which enabled the team to discuss and make decisions concerning the project as it progressed. Tasks that were to be completed soon were discussed in-depth, whereas tasks scheduled for later dates were discussed in a more abstract, high-level manner. Our team was composed of individuals who had the necessary knowledge of the project parameters, hardware and software used in the project to make educated estimates for the projected times. When tasks were completed and it was time to analyze the subsequent tasks, dates may have been negotiated and modified depending on the coming in-depth discussions and knowledge of the team. The following diagram and sections present the process (Figure 9) and project schedules (Figure 10, Figure 11 and Figure 12) our team utilized over the past two semesters, respectively.

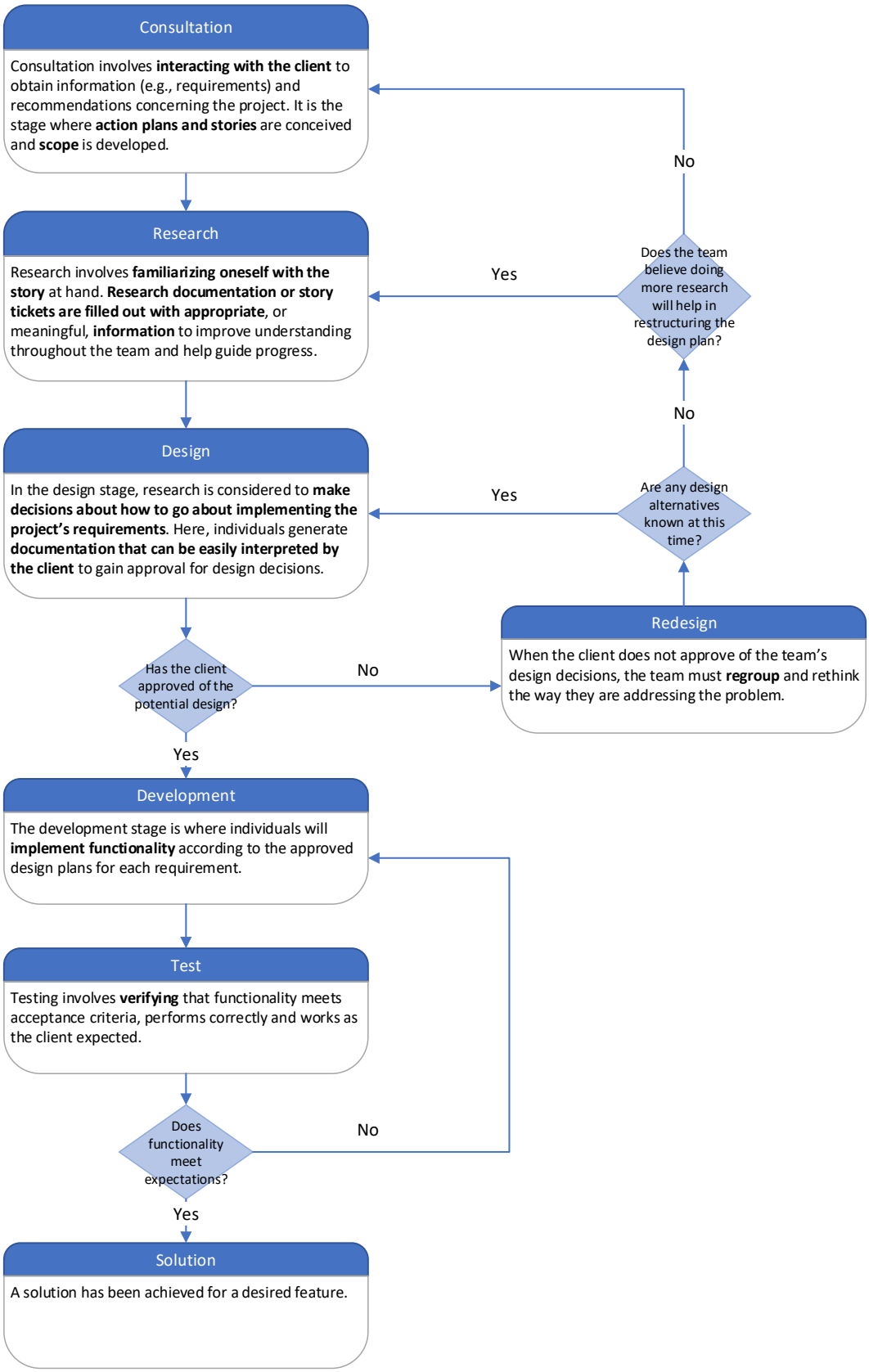
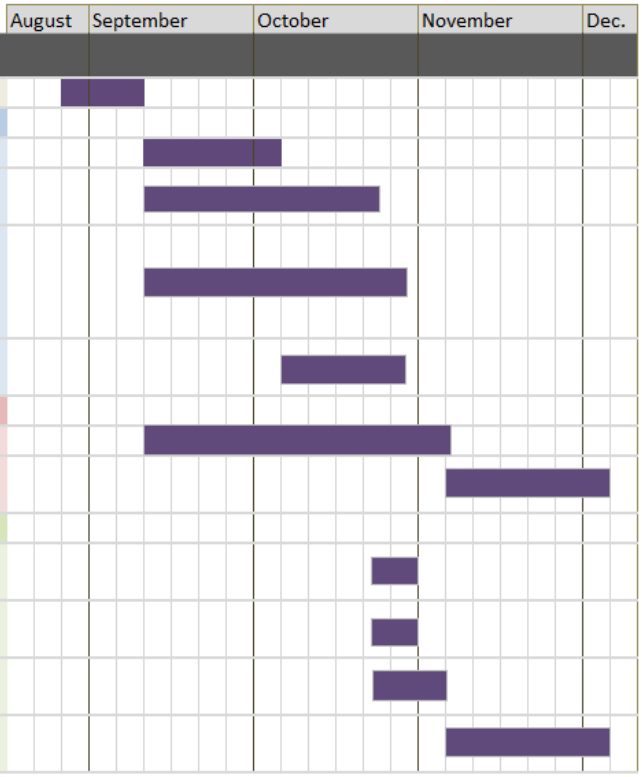


FIGURE 9 PROCESS FLOW DIAGRAM

Proposed Schedule

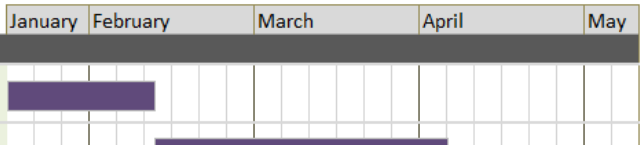
Project Schedule Fall 2018



TASK	START	END
Initial Bitlocker Consultations	8/30/2018	9/6/2018
Research Phase	**9/10/2018**	**10/23/2018**
Familiarize team with PUF	9/10/2018	10/8/2018
Determine if and how encryption can be performed at kernel level	9/10/2018	10/23/2018
Familiarize team with PUF applications created by previous senior design teams (reading and testing code and client consultation)	9/10/2018	10/29/2018
Create documentation summarizing findings	10/8/2018	10/29/2018
Design Phase		
Establish an initial design document	10/8/2018	11/5/2018
Improve design document, establishing a more official architecture	11/5/2018	12/7/2018
Development Phase		
Eliminate unnecessary code from provided PUF repository	10/22/2018	10/29/2018
Create a new Android application for the bitlocker (PUF-based)	10/22/2018	10/29/2018
Fix broken algorithms within the provided PUF library	10/22/2018	11/5/2018
Implement application level encryption on Android	11/5/2018	12/7/2018

FIGURE 10 PROPOSED FALL 2018 SCHEDULE

Project Schedule Spring 2019



TASK	START	END
Implement kernel level encryption on Android	1/14/2019	2/11/2019
Identify and integrate other functionality desired by the client	2/11/2019	4/8/2019
Testing Phase		
Identify and resolve all outstanding bugs	3/25/2019	4/15/2019
Make improvements or additions as needed (technical debt)	4/8/2019	4/19/2018
Project Delivery	4/29/2019	5/3/2019

FIGURE 11 PROPOSED SPRING 2019 SCHEDULE

The schedules presented in Figure 10 and Figure 11 were proposed at the beginning of the fall 2018 semester. To formulate our schedule for the fall and spring semesters, our team decided to organize our project into five phases: research, design, development, testing and delivery. During the research phase, our team was expected to investigate concepts essential to

developing our encryption application, which, in turn, would lead to the establishment of an appropriate design plan to guide the team in the development phase.

Since our team used the rolling wave planning technique, the proposed fall 2018 schedule became more detailed compared to the spring 2019 schedule since there were too many unknowns about where we would be in terms of progress at the beginning of the spring semester. We realized that the spring schedule would become more elaborate as we learned more information about the project. Consequently, the spring schedule simply outlined fundamental goals we hoped to attain.

Actual Schedule

Regarding the fall 2018 schedule (Figure 10), our team did not stray too far from the items outlined in each phase. The team decided to spend the first two months getting familiar with the concept of a PUF, encryption techniques and the PUF applications provided by our client. Doing so ultimately assisted in creating an appropriate design plan for implementation. As the research and design phases progressed, our investigations unfortunately showed that the library used by the PUF applications contained errors. As a result, the team agreed to establish a few milestones to be achieved within the first part of the development phase, which included furthering our understanding of fundamental concepts required for implementation of our client's desired application, fixing the broken algorithms within the provided PUF library and implementing a working encryption application using PUF at the application level.

As predicted in the initial schedule planning stage of our project, the spring 2019 schedule was reworked multiple times. Below is the schedule we agreed to follow during the spring term:

Project Schedule Spring 2019

TASK	START	END
Research Phase		
Determine feasibility of full-disk encryption	2/18/2019	2/25/2019
Investigate extension of application as lock screen	3/25/2019	4/8/2019
Determine feasible methods of kernel level encryption	3/25/2019	4/15/2019
Development Phase		
Locate and resolve PUF-related issues	1/21/2019	4/17/2019
Modified normalization method	1/21/2019	1/28/2019
Implement key generation from quantize data	2/4/2019	4/8/2019
Implement application-level encryption by user profile	2/14/2019	4/8/2019
Integrate PUF library features with PUF application	4/8/2019	4/15/2019
Testing Phase		
Integration of newly written java libraries (interpolation pressure)	2/4/2019	2/25/2019
Testing normalization method	3/25/3019	4/8/2019
Testing encryption	3/25/2019	4/15/2019
Testing key generation	3/25/2019	4/28/2019
Integration testing of final iteration of product	4/18/2019	4/28/2019
Project Delivery	5/1/2019	5/1/2019

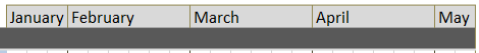


FIGURE 12 PROJECTED SPRING 2019 SCHEDULE

At the beginning of the spring semester, our project underwent a scope change where fixing the inaccurate PUF library became a priority. Once we resolved enough issues with the library, the team began to spread out into different areas. Some members began implementing new features, such as key generation and application-level encryption, others re-entered the research phase to investigate areas essential to achieving our updated functional requirements, while the remainder continued work on resolving bugs in the PUF library. As more features were completed, our team slowly began integrating our test plan to validate and verify feature functionality worked as the client expected.

6.3 Potential Risks

There were several challenges that accompanied the inheritance of a four-year-old library. One observation made was a heavy use of hard-coded values, which made updates and patches hard to perform. Naturally, fixing issues is much more manageable than creating a brand-new solution. The lifetime of the library had also resulted in multiple implementations of normalization, key generation and authentication. The team believed this could have been a blessing or a detriment to the project. In other words, we thought having these various solutions could have made it easy to switch between implementations based on needs at a given time, or they could have made it difficult to create a functioning program, as piecing together different parts present a challenge. In our case, it turned out to be somewhat difficult to piece together the correct algorithm.

Another risk the team identified was the level of encryption that was possible to implement within the Android SDK. It was unclear whether we could implement encryption at the kernel level. It was also unclear if implementing full-disk encryption was a feasible solution for this application or whether it would create an unfriendly user experience by 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. 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.

6.4 Setbacks and Mitigation

6.4.1 PUF Library Issues

As expected, the team experienced issues when interacting with the PUF library. The library was not in working condition when the project began; it constantly output false positives and negatives. This warranted modifications to be made to the library because the solution required a high level of accuracy and authentication.

To mitigate this issue, the team consulted our technical advisor, Timothy (Tim) Dee. Tim was able to provide insight and specific plans of action from his knowledge and experience with the library. The library required modifications with respect to how challenges were drawn on the screen, which normalization method show be used and how authentication should function.

A major issue that took time to fix was the way in which normalization needed to change. The version of the library the team was given used a Java library that interpreted Python to utilize the Python scripts needed for normalization. The project structure changed in a way that caused the Python scripts to be interpreted incorrectly, causing the normalization results to be incorrect as well. The solution to this was to remove the need for the Python interpreter by converting a Python script, “Util.py,” to Java. This made the repository more uniform, comprehensible and reliable by making it more Java dominant and less dependent on a third-party for core functionality.

6.4.2 Encryption Level

The team was unable to attain the original goal of kernel-level encryption. The Linux kernel utilizes a library, “fscript,” to achieve filesystem-level encryption by encrypting files and directories. The team met development issues when interfacing with the Linux kernel, ultimately deciding to adjust the level of encryption the application would implement.

To mitigate this issue, the team decided to implement application-level encryption. This would still obfuscate user data but not occur on the file system directly. Application-level encryption would also allow for a more user-friendly experience and mimic the functionality of a password protected application. The user would only see the data decrypted when authenticated inside the application, and the data would be encrypted once the user closed or left the application.

6.5 Lessons Learned

Throughout this senior design project, we learned a lot about what it means to work on a team and how to execute various aspects of the project lifecycle within the context of a large-scale project. We made many mistakes throughout our project, but these blunders were instructive

as we learned what helped make our team be more effective and efficient. Our experiences, although frustrating at times, were valuable in helping our team grow as developers and prepare for the working world after graduation.

Over the course of the project, facilitating communication to maintain accountability and exhibiting strong work ethic had been recurring issues for our team. Regarding the former, we struggled to convey when assigned work was expected to be realistically completed when considering an assignee's personal schedule and circumstance. Many members of the team had swamped schedules for extended periods of time, technical issues with the devices they were working with or unforeseen events to work through, and a failure to communicate these issues early on resulted in many deadlines being extended. Realizing what was happening late in the fall semester, we attempted to implement an accountability plan where members would reach out to other members via Group Me (i.e., a group chat application) to check in on progress whenever meeting in person was not an option. This helped all members be more aware of what was happening with a feature and made it easy to determine if help was required on a story. Although it was unfortunate these events occurred, they are very real problems that exist in the working world. Learning to mitigate our communication issues during our academic careers is beneficial knowledge that will become extremely advantageous when working with professional development teams.

In terms of the latter idea of worth ethic, our team realized many of the circumstances outlined above were justifiable in that individuals may not have been able to work; however, instances of "laziness," or lack of motivation to work when there was enough time available, were also evident. This issue is another real problem many organizations face in the professional world. An individual cannot force another individual to do work; however, incentives can be placed to encourage better performance. In the case of our team, we found that acknowledging the work others have done and organizing group work sessions helped in facilitating progress. These forms of incentives may not work for every team but knowing they are viable options can help expedite the mitigation process.

In addition to learning how to operate within a team, we learned how to work through all phases of the project lifecycle while handling difficult situations. From our previous work and academic experiences, we were familiar with how each phase should be executed; however, we struggled with where to begin in each phase. We initially only had vague ideas of what we were doing, so we took steps to become familiar with the concepts and content we would be working with. Unfortunately, upon exploring one of the applications our client provided, we discovered that the library we received did not function the way the client claimed it did. This event occurred after months of researching and planning for the implementation of our client's desired features, so learning that the milestones we had been accounting for might not be attainable caused discouragement to flow through the team since we felt like we wasted time.

This scenario is, yet again, another real problem an organization may face when requested by a client to build upon an existing piece of software. If it is known that a team is expected to expand upon a preexisting piece of software, failing to evaluate a provided product's functionality to determine if the results align with the client's claims could result in a lot of wasted time when planning and designing the next steps, especially if the results of the evaluation are unexpected. Despite our frustration, we were able to backtrack through our project workflow to reevaluate our scope and requirements and then push forward into the development phase with a new plan of action. Experiencing this type of scenario helped our team understand that we should always explore a product first to verify that the product functions as expected and then plan, design and develop the next steps accordingly. Doing so will save a team from many headaches and feelings of discouragement.

7 Closing

7.1 Conclusion

Overall, the purpose of this project was to create a more secure way to protect data on a user's phone. This project was 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 is encrypted by using Android's encryption API. As we approach the close of our project, we can say we have created a mostly successful application that will encrypt and decrypt files using the PUF library's response.

7.2 Going Forward

The objective of the client was to implement his theory of pressure-based authentication into all phones at the kernel level. To do so, we believe there are several potential steps forward. Considering our research, we found that implementing our application at the kernel level could take the place of implementing a lock screen. Android provides a library that could be extended to create a lock screen of sorts; however, the library is not well-documented and would require a substantial amount of work to develop. This implementation would only work with Android devices. Consequently, if the client wanted to incorporate pressure-based authentication in all phones, the idea would have to be pitched to a cellular company and obtain that company's compliance.

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.