5 Testing

5.1 Unit Testing

Our CRC-16/KERMIT checking class will be tested for accurate packet validation. This will be tested by using a distribution of valid and invalid packets to give the class. CRC validation needs to operate at 100% accuracy, so evaluating accuracy is actually rather simple.

Inside our AWS infrastructure, ensuring that the data is passed between services is crucial. By running basic test payloads in the AWS console, we can see if the data is successfully passing through. Our packet analyzer lambda needs to receive information from the IoT Core through the SQS so we can run our mock basestation script to send information to the IoT Core. From there, using CloudWatch to view the logs, we can see whether the data reaches the analyzer lambda successfully or what errors may have occurred. As well as testing in the cloud, these scripts can be tested locally to see how they react because we know what the packet format will look like.

Additionally, our front-end application will incorporate unit testing to ensure the accuracy of what we are designing. The Flutter framework has a nice directory in the main application directory to create and run unit tests called /test. All that is needed to do to be able to run unit tests is to add the test or flutter_test dependency to the pubspec, package manager file. To add even more functionality to our unit tests, we will also use the Mockito dependency. With unit testing enabled for our front-end application, we will be able to make sure data can trigger certain functionalities and features in our app. For example, we could mock up a json value and make sure the mocked json is displayed correctly on the screen and returns the correct response code. For every functionality of our application, we will include a unit test to verify the correctness of the functionality.

5.2 Interface Testing

Our two biggest interfaces for this application would be our IoT sensor network and our AWS infrastructure. Due to the interconnectedness of these two interfaces, it will be essential to not only test them in their own unit test but also test them together as one large interface. These tests will include reverse engineering our basestation code and attempting to send malicious or malformed packets to it from our sensors. If we are able to get one of these modified packets through the basestation it will be put into our AWS database and could potentially cause more damage to the stability of our application.

Just as we test the dataflow from our IoT sensors to AWS, it will be important to understand the ways in which data from AWS can affect our sensor network. We will be attempting to implement a naming system where users of our application can name and digitally place sensors over a map image of their land to better keep track of the positions their readings are coming from. This feature will require data to be sent from our user configuration database

in AWS to the sensor and interact with them. This will require a large amount of manual testing to ensure AWS is not able to put the sensors in some state that is unrecoverable or requires physical interaction with the device (hard resetting it).

5.3 Integration Testing

One significant integration path on the hardware end of our IoT system is Sensor-to-Base Station communication. Due to constructing our own base station, a lot of testing will need to be completed on both communication paths of the base station to ensure our implementation is functioning properly. Due to the nature of the LoRaWAN protocol, we can expect about a 10% packet loss rate as reasonable for our system and its applications. Communication testing outside with obstacles (or the actual crops they're meant to simulate) should be completed for reliability. If our packet loss rate is consistent with the 10% margin, it will be considered suitable for our agricultural application. Since the packet structure created by our sensors have a CRC code attached to it, accurate transmission can also be examined.

The next integration path is between the base station and the AWS IoT core. This will operate over the TCP/IP stack and use an MQTT application protocol. Reliability testing will need to be completed at this stage to evaluate the functionality of our MQTT implementation. Again, due to constructing our own base station, testing will be more focused on the base station and its configuration with the AWS IoT Core. Due to our IoT system's infrequent data collection, speed and latency are not as important to consider during testing. MQTT (particularly Quality of Service 2) is considered very reliable, and our pathway will need to be held to that standard. To test this, we can manually disrupt the network connection of the base station and evaluate how it recovers. If, after the disruption, the base station still accurately sends its data and it is properly received by the IoT core, then our system can be considered reliable for data transmission to the cloud.

The final integration path will be between the AWS API gateway and our Flutter front-end application. This connection will be made possible through a Rest API gateway provided by AWS and received by our front-end application. To test the functionality and correctness of this connection, we create integration tests in our front-end application. These tests will be similar to how our unit tests for our front-end application will be implemented. We will add the Flutter integration_test dependency to our pubspec and after that, it will be as simple as running the tests we create to see if they pass. This will allow us to test the functionality of our application to see functionality like if buttons are being triggered correctly and pages are being displayed properly.

5.4 System Testing

Our full loop system consists of three smaller subsystems in the IoT sensors, AWS infrastructure, and frontend application. For system testing, we will need to ensure that we can have data transferred through our entire system. Once we can do that, we will be able to see how the system interacts as a whole. This form of testing will be manual in nature because of the size of our system. Each form of testing prior to this will be integral to getting a foundation for our system testing. Unit testing will allow us to look at a specific aspect of our system, interface testing will help us see how two parts connect to each other, and integration testing will show us how those paths of dataflow in our system move.

5.5 Regression Testing

Since many of our development changes will be centered on the base station, consistent regression testing to determine that new security additions have not broken our sensor-base station-cloud communication path for uplinks and downlinks. A few uplink and downlink transmissions should be sufficient for smaller modifications and can be expanded for larger design improvements for more thoroughness.

The next way our project will regression test is by creating a Continuous Integration and Continuous Deployment pipeline. This will allow us to test our new code on our old unit and integration tests. We will create these CI/CD pipes in Gitlab, and once they are set up, they will run every time we push our code to Gitlab. If one of our old tests fails, the CI/CD pipeline will also fail, indicating to us that there has been a regression. This will help us with the AWS side of our project as well as in our front-end application.

5.6 Acceptance Testing

It will be important for us to have a document that keeps track of all the test cases we have been running. We will keep track of whether the test was successful or not and, if not, what specifically the error was. In order to monitor which areas of our project we have tested we will categorize the attacks and mitigation techniques based on the relevant area of the project that they apply to such as, IoT devices, AWS infrastructure, or frontend application.

To make sure that our client is aware of our testing progress, we have been providing live demonstrations during our weekly meetings. Going forward, we will continue this but also increase communication so that our client is always aware of our progress. Our client has stressed that for a successful project we should ensure that the data communication is sent securely from the sensors to the user through Amazon Web Services. To make sure that our clients' needs are satisfied we focus not only on securing the inter-component communication but also securing data inside each component such as information stored in databases or adding more sensors.

5.7 Security Testing

Significant security testing will need to be completed as a part of our project's objective. For Sensor-to-Base Station communication, three key attack vectors will need to be examined: packet sniffing, packet modification, and imposter sensors introduced to our system. Packet sniffing can be mitigated by the standard LoRaWAN protocol's use of AES-128 bit encryption. Verification testing can be done to ensure all packets are properly encrypted with AES-128. Packet modification can be mitigated by the aforementioned AES-128 bit encryption and the CRC-16/KERMIT verification. If a packet is modified, the base station will be able to verify the validity of the data by calculating and comparing the CRC code for each packet. This can be done by passing a mix of accurate and modified packets to the base station to track its ability to evaluate data validity. Imposter sensors are trickier to test. LoRaWAN specified Over the Air Authentication (OTAA) as the join procedure. OTAA is already well established as a protocol so testing will revolve around the implementation of the protocol rather than its validity. To test this, we can introduce an unregistered sensor into the range of the base station and test whether it will be able to connect/authenticate with the base station without our approval or knowledge.

There will also be significant security testing for the connections between our front-end application and our AWS infrastructure. This will mostly include manual penetration testing of our mobile application and our API. We will use industry-standard tools BurpSuite and SQLMap for testing our login system. This round of testing will ensure proper mitigation of SQL injection and authentication bypass. After this, we will also use Postman and BurpSuite to see exactly what data our application is processing. We will then use this data to test our APIs and ensure proper access controls are being enforced on our users.

More security testing within our AWS environment will involve role-based access control (RBAC). This will ensure that our frontend users will only be able to access data specific to their users and not escalate privilege. Though we have not made a final decision on data storage, RBAC will make sure that the front-end can only see the correct data. Also, by ensuring that users can't switch roles if they were to access our environment, we can prevent unauthorized edits to our infrastructure.

5.8 Results

Testing our system will ensure that we meet our standards and requirements for functionality and specifications. With each testing variant, we will be able to verify the validity of another part of our system. For unit testing, we will verify that data packets are being sent properly, AWS can read and send data, and that the data can make it to the front-end application. For interface testing, we will focus on ensuring that our IoT sensor network interface is tested together with our AWS infrastructure interface. For integration testing, we will focus on our integration paths of sensor to basestation communication, basestation to AWS IoT Core communication, and AWS API gateway to our Flutter front-end application communication. For

system testing, we will focus on a more manual form of testing similar to QA to help us find if there are faults in our overarching system that need to be addressed. For regression testing, we will set up a Continuous Integration and Continuous Deployment pipeline to verify that previously written unit and integration tests do not fail when new code is written. For acceptance testing, we will create a document that lists all of our tests with information on whether they were successful and a description of the problem that is being addressed. Finally, for security testing, we have three distinct security areas. The first area will be sensor-to-basestation communication, where we will focus on protecting our system against packet sniffing, packet modification, and imposter sensors introduced to our system. The second area will be between our front-end application and AWS infrastructure, where we will mainly focus on penetration testing. The final area we will focus on will be in our AWS infrastructure with role-based access control.